Interface
2003 January

［表紙デザイン：㈱プランニング・ロケッツ］

Contents 2003 January

特集
オリジナルアーキテクチャのパソコンを作ろう！

Interface

Contents 2003 January

話題のテクノロジ解説

ショウレポート&コラム

一般解説&連載

■情報のページ

連載「開発環境探訪」「やり直しのための信号数学」は,お休みさせていただきます.

国内最大のエレクトロニクス総合展

CEATEC JAPAN 2002

北村俊之

「ブロードバンドの先に，次が見える.」をテーマに「CEATEC JAPAN 2002」が10月1日(火)～5日(土)の5日間，日本コンベンションセンターで開催された．主催は情報通信ネットワーク産業協会(CIAJ)，(社)電子情報技術産業協会(JEITA)，(社)日本パーソナルコンピュータソフトウェア協会(JPSA)である．CEATEC JAPANとしては今年で第3回目となる本展示会は，通信，情報，映像分野の最先端技術，製品，サービスが一堂に会した，業界をあげてのアジア最大級の複合展示会となっている．最終的な延べ来場者数も173,021人となっている．

〔写真1〕会場のようす

例年どおり非常に大規模な開催となっており，全体が「電子部品・デバイス＆装置」「ビジネス・ソリューション」「ネットワーク・ソサエティ」「ホーム＆パーソナル」の四つのパートに分けられていた(**写真1**)．

● 電子部品・デバイス＆装置

日立製作所は，「携帯電話を加速する」をテーマに，携帯電話向けアプリケーションプロセッサ「SH-Mobile」フラッシュマイコン「F-ZTAT」シリーズ，モバイル/ネットワークやカーエレクトロニクスなどの各種ソリューション，フラッシュカード，汎用半導体などの製品を展示していた(**写真2**)．ミツミ電機は，マルチファンクションタクティールスイッチ「SOQ Series」，光コネクタ「MC-901/903/918」，Bluetoothモジュール「WML-C09/C10/C11」を中心に展示を行った．「WML-C09/C10/C11」は，Bluetooth Class1/Class2に対応し，UART/USB/PCMをインターフェースにもつことで，幅広いアプリケーションを実現するとのことである．

〔写真2〕SH-Mobileを使用した指紋認証システム

〔写真3〕シャープの3D液晶ディスプレイ

シャープは，ディスプレイ，ネットワーク，システムの各種ソリューションデバイスの提案を行っており，3D液晶ディスプレイや独自の無線，光技術を使用した製品に来場者の注目が集まっていた(**写真3**)．ナショナルセミコンダクタージャパンは，“Powering Innovation”をテーマに，「パワーIC」「映像の世界」「音声の世界」「コミュニケーション」の各コーナーで最新半導体/半導体技術のほか，同社製品を採用した各社製品の紹介デモを行っていた．

● ビジネス・ソリューション

ナカヨ通信機は，無線LAN画像伝送システム，高速100Mbps光無線システム，次世代IPビジネスホンを中心に展示を行っており，IEEE02.11技術を用いたワイヤレス動画像伝送システムや次世代型のIPビジネスホンは，来場者の関心も高いとのことであった(**写真4**)．

〔写真4〕ナカヨ通信機のIPビジネスホン

NECは，CPU，LSI，メモリ，集積回路，サーバからインターネット関連，ビジネス関連アプリケーション，IMT-2000，ブロードバンドなどの各種ネットワークサービス，さらに携帯電話，PHSなどの各種通信サービスなど幅広い展示を行っていた．沖電気工業は，CTIシステム，インターネット/イントラネット関連機器，VODサーバなどの製品を展示しており，とくにIP電話サービスプラットホーム，VoIP評価/検証ソリューションは，IP電話への関心の高まりとともに注目度も上がっているという．また，CTstage 4iは，プラットホームにMicrosoft.NETを採用し，インターネットと親和性の高いSIPやIPv6に対応，通信系と情報系システムを統合する高付加価値ネットワークインフラを実現するという．

● ネットワーク・ソサエティ

NTTドコモは，FOMAの最新端末「FOMA T2101V」やPDA一体型タイプの「FOMA SH2101V」などの展示を行っており，来場者の高い関心を集めていた．またユビキタス社会の実現に向けた，さまざまなモバイル技術を活用した取り組み事例を紹介していた(**写真5**)．

〔写真5〕FOMA・ペット型ロボットコラボレーションシステム(参考出品)

KDDIは，位置情報提供サービス「KDDI GPS MAP」が注目を集めていた．このサービスは，携帯電話とパソコンだけでGPS位置情報サービスが利用できるというもので，同社のセンタープッシュ機能を採用することで，センター側からのリアルタイムな位置検索が可能であるという．

〔写真6〕東芝のGENIO e550G

東芝は，FOMA，au，TU-KA対応の携帯電話各機種のほか，PDAである「GENIO e」などモバイル端末に人気が集中していた．とくにPDAでありながら，4.0型の液晶画面を搭載し，Intel PXA250-400MHzアプリケーションプロセッサを搭載した「GENIO e550G」に力を入れているとのことであった(**写真6**)．

三菱電機は，日本最小級のサイズを実現したETC「EP-400」シリーズの展示を行っており，同製品は小型で外付けスピーカ，車内の好きな場所に置けるのが特徴だという．J-PHONE対応の「J-D06」はモバイルカメラ，日本語変換ソフト，着信相手を画像で知らせる，3Dポリゴンエンジンなど多彩な機能を搭載した製品である．

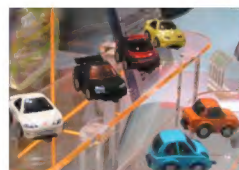
● ホーム＆パーソナル

パイオニアでは，高品位ディジタル信号を高速，高精度に伝送可能なディジタルインターフェース「i.LINK」を搭載したマルチチャネルプリメインアンプ「VSA-AX10i-N」およびDVDオーディオ/ビデオSACDプレーヤ「DV-S858Ai」の展示を行っていた．このi.LINK規格の端子をもつ機器同士であれば，DVDオーディオやSACDなどの高音質ソースをダイレクトに伝送することができる．

ソニー/ソニーマーケティングは，地上波からデジタルハイビジョン放送まで，さまざまな映像信号を，ブラウン管，PDP，液晶などすべての映像表示デバイスにおいて高画質映像を実現する新開発の統合デジタル高画質システム「ベガエンジン」を搭載した“ベガ”4シリーズ計9機種やau対応の携帯電話の着せ替えカバーなどを展示していた．メモリースティックが搭載できる「ミュージックチョロQ」(タカラ)など，遊び感覚にあふれた展示も多く見られた(**写真7**)．

〔写真7〕タカラのミュージックチョロQ

Show report

Show&News Digest

WPC EXPO 2002

■日時：2002年10月16日(水)～19日(土)
■場所：東京ビッグサイト(東京都江東区)

ブロードバンド時代――ユビキタス・ネット社会を拓く――と題して，パソコンをはじめとしたコンシューマ機器の展示会が開催された．国内最大規模の展示会ということもあり，来場者は363,590人に達した．

今回のWPC EXPOでもっとも目についたのはTablet PCである．キーボードレスで液晶ディスプレイを搭載し，持ち運びが可能なペン入力PCで，マイクロソフトのWindows XP Tablet PC Editionをはじめとして，各社のTablet PC製品が展示され，来場者の大きな注目を集めていた．

もう一つのキーワードは「ホームサーバ」だった．東芝の「ワイヤレスホームメディアステーションTransCube」はブロードバンドルータ，無線LAN，ハードディスクビデオレコーダを統合化したソリューションである．一家に一台のホームサーバ時代の到来を感じさせる製品である．

IBMでは，同社のノートPC，ThinkPadシリーズの10周年を記念し，歴代モデルを展示していたほか，記念モデルの予約受け付けなども行っていた．

そのほかにはソニーテクトロニクスから社名変更した日本テクトロニクスのBluetoothプロトコルアナライザBPA105型が展示されていた．従来製品と比較してトリガ機能が強化され，指定のイベントやトランザクションだけをとらえ，記録・表示が可能になった．

IBMの歴代ThinkPadの展示

東芝のTransCube

マイクロソフトのブース

Tablet PCの展示

日本テクトロニクスのBluetoothプロトコルアナライザ

FSIJ国際シンポジウム

■日時：2002年10月22日(火)～23日(水)
■場所：日本教育会館(東京都千代田区)

7月に発足したNPO，フリーソフトウェアイニシアティブ(FSIJ)による国際シンポジウムが開催された．開催されたセッションは「中国におけるフリーソフトウェア」，「Free Standard Groupセッション」，「Debianセッション」，「ヨーロッパにおけるフリーソフトウェア」，「ヨーロッパ，アジアにおけるフリーソフトウェア」など．

Martin Michlmayr氏によるDebianソフトウェアの品質管理に関するセッションでは，世界各地に点在するメンテナの動向を把握する必要性と，そのためのデータベース作りなどについて語られた．Neal H. Walfield氏によるGNU Hurdセッションは，GNU Projectで開発が行われているOSであるGNU Hurdについて，従来のOSと比較して速度は劣るものの，マルチサーバアーキテクチャによりセキュリティが強化されることなどが解説された．

セッションのようす

ARM，Derek Morris氏へのRealViewに関するインタビュー

■日時：2002年10月21日(月)

ARMのDevelopment Systems General ManagerであるDerek Morris氏に，同社の新製品RealViewに関してインタビューを行った．

編 まず，RealViewについて簡単に説明してほしい．

Morris RealViewはARM向けのデバッガだ．ARM11をサポートしているほか，マルチコア混合アーキテクチャ――複数のARMコア，複数のDSPコアのデバッグが同時に行える．複数のコアに対してSTOP/RUN/STEPが可能だ．

編 複数コアを用いた開発が増えているのか？

M 2000年の調査では，デザイン数ベースで，複数のARMコアを用いたデザインが2%，ARM+DSPが8%，ARM+コプロセッサが9%という統計がある．また，次のデザインでは，64%が複数コアを用いた設計を行うという．

編 RealViewの製品構成について教えてほしい．

M RealViewはコード生成ツール(RV-CT)，RealViewデバッガ(RVD)，RealView ICE(RVI)などの単体売りを行う．顧客は必要なものだけを購入することができ，選択の幅が広がったことも特徴だ．

Derek Morris氏

自分自身を語るオブジェクト指向「物」

山本　強

最近のソフトウェア開発のスタイルといえば，オブジェクト指向である．オブジェクト指向の優れたところはたくさんあるのだが，基本となるのは「オブジェクト」という単位であり，そしてそれが何であるかを自らが知っている，知ることができるということではないかと思う．

元祖オブジェクト指向言語であるSmalltalkは，言語仕様がオブジェクト指向的であるということがもちろん重要な要素なのだが，開発環境として個々のオブジェクトの機能や定義を簡単に参照できる仕掛けが含まれていることにも大きな意味がある．今では，ほとんどの人がインターネット用語だと思っているブラウザ(Browser)も，もともとはSmalltalkのオブジェクト閲覧ソフトウェアの名称だったのである．

そこで今回は，ソフトウェア設計にこれだけ大きな影響を与えたオブジェクト指向を，もっと他の分野にも適用できるのではないかと考えてみた．もっとも，それは自然な考え方ではあるが．

自分を語る「物」

最近になって製造責任者や産地証明といった「物」に対する説明責任が高度に求められるようになっている．すなわち，工業製品や農作物について，それがどのようにして作られたものなのかを消費者が知る権利が確立しつつある．今のところ，消費者側の欲求は生産者や流通業者に対して嘘をつくなという精神論的な要求なのだが，いずれ技術的な証明が求められるようになるであろう．

こんな話もある．携帯電話を買うとマニュアルが何冊もついてくる．携帯電話の本体は100gもないのに，その数倍もある分厚いマニュアルが何冊も付いてくる．ほとんど読まれないと思うが，それが必要になったときにはどこかに片付けられているか，捨てられてしまって見当たらないということが多い．

最近のIT家電製品は，どれも似たような状況になっている．かつて，OAブームで盛り上がったとき，オフィスから紙が消えるとまことしやかにいわれたが，実際には紙の消費量が爆発的に増加したという経験がある．形は違っても，今また同じことを繰り返しているように見えるのは筆者だけではないと思う．

また，ITを生業とする人にとっても，ハードウェアとデバイスドライバの分離というやっかいな問題がある．ハードがあってもデバイスドライバがなければ使えないのに，OSはアップデートされてもデバイスドライバはないということが多い．プラグ&プレイは製品の情報まで語ってくれるのだが，デバイスドライバへのアクセスパスまでは語ってくれないからはがゆい．こういう問題は，「物」が自分をしっかりと語るしくみをもってくれれば解決する．

バーコードから始まる「物」のオブジェクト指向化

工業製品のほとんどにバーコードが印刷されるようになって久しいが，バーコードも拡大解釈すると「物」のオブジェクト指向化ツールと考えることができる．製品がインスタンスであり，それを生み出す製造ラインや規格がクラスとなる．バーコードは製造工場と製品規格へのポインタであり，それが数値としてバーコードにコーディングされていると考えることができる．今のところ，それを見るためのブラウザが消費者に提供されていないので，オブジェクト指向のメリットが見えないのである．

また，バーコードもどんどん進化している．2次元バーコード(QRコード)による大容量化や，JRのSuicaに採用されているRFID(無線タグ)による非接触化などがすでに実用化されている．印刷型はリードオンリーだが，RAMを内蔵した書き換え可能なタグも低価格化している．そして，その先にあるのが，スマートダストと呼ばれる超小型無線LSIタグであろう．スマートダストというコンセプトは，それこそ印刷インキに混入できるレベルまで微小化した書き換え可能なタグである．

「物」が自分を語るしくみ

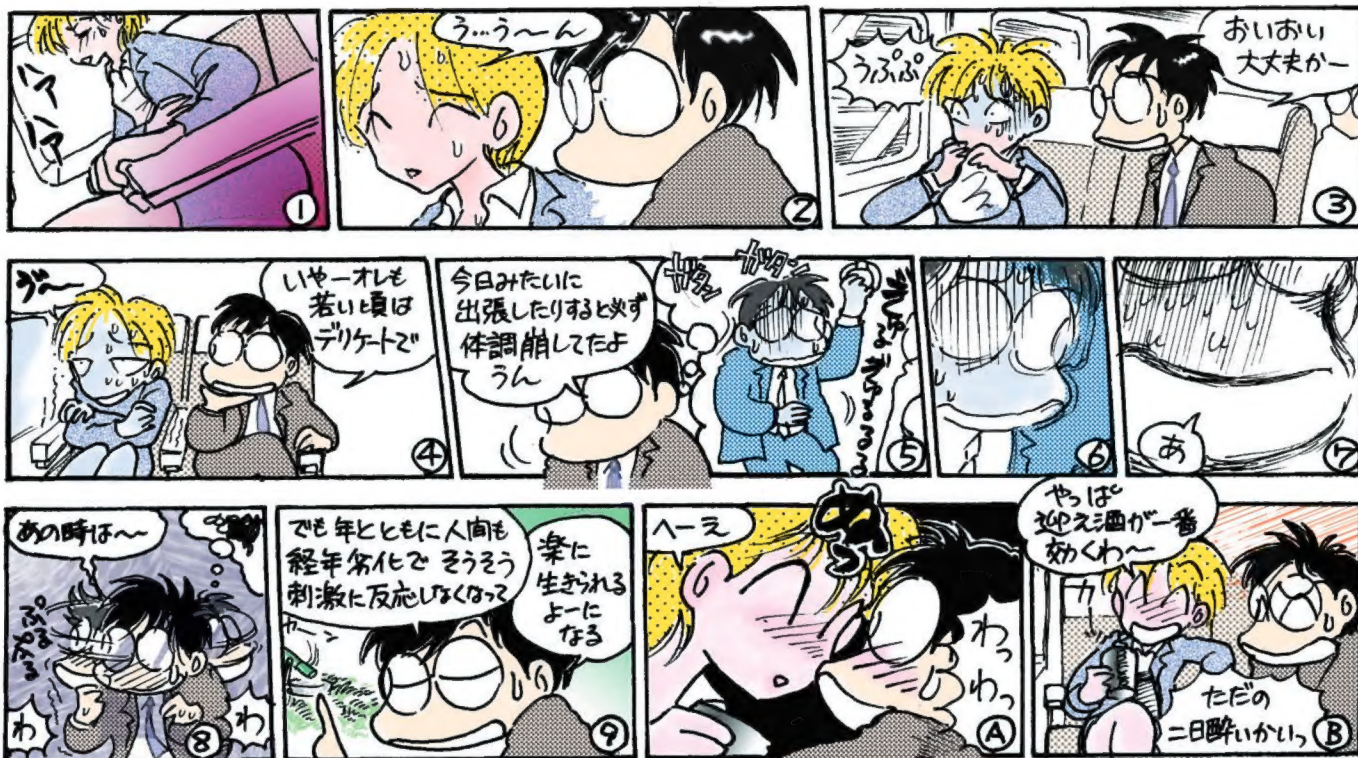
バーコードが製造規格へのポインタであると考えると，ポインタの先を見えるようにするのが「物」のオブジェクト指向化の第一歩である．ネットワークが遍在化した今なら，クラス情報をネットワーク経由でアクセスするのがいちばん簡単である．幸い，日本は世界に誇れる移動体通信インフラをもっている．農作物は工業製品ではないが，生産者情報や生育過程がクラス情報となる．これは，書き換え可能タグとネットワークを経由した情報提供が実現できることになる．

さらに，タグの大容量化と標準化という作戦もある．CD-ROM並みの情報が入るタグができるなら，クラス情報をすべて入れてしまうことができる．すなわち，仕様やマニュアル，基本ドライバなどはタグにすべて刷り込まれているという状況である．

そこで問題になるのが，ドキュメント構造とブラウザの標準化である．これが実現できれば，一家に一台のマニュアルリーダがあればどんな「物」であっても，そのマニュアルが読めるようになる．そうなれば，今度こそ紙とプラスチックの消費量が減少するのは間違いないのではないだろうか．

＊　＊　＊

「ユビキタス」が流行語になっているが，思想のレベルで何が変わるのかという説明が明確ではないように見える．ちょっと便利になるという程度の説明では，インターネット冷蔵庫は売れない．すべての製品がオブジェクト指向でいう「物」になるというのが，ユビキタス社会の実現イメージかもしれない．

やまもと・つよし
北海道大学大学院工学研究科電子情報工学専攻
計算機情報通信工学講座　超集積計算システム工学分野

フジワラヒロタツの現場検証(66)

歳を重ねるということ

パタパタと仕事に追われているうちに夏も過ぎ，おや秋の気配と思っているうちに，もう初冬を思わせる寒さに身を震わせていたりします．筆者の会社は古いビルに入っていて，空調は真夏と真冬にしか効かないので，会社に居ながら季節感を感じることが容易という利点があります．たぶん，もう少しちゃんとしたオフィスビルに入ってらっしゃる方には，季節の移ろいはなかなか感じにくいものではないでしょうか．大きいビルは，人工都市みたいなものですし．

もっとも，古くても新しくてもどちらも蛍光灯の下，ずっと人工光で暮らしていれば同じかもしれませんね．筆者も季節感を感じやすいビルなどといいつつ，ディスプレイに光が映り込まないように朝からブラインドをおろして仕事をしています．

まあ，季節や昼夜を考えずに働いていても，自ずと歳はとるものですね．かつて読んだノンフィクションに「コンピュータ新人類の研究」という本があって，「こりゃまるで俺のことが書いてある！」などと感じた，いわゆる第1期マイコン少年の筆者も，とうとう不惑の年になってしまいました．

で，年をとって感じるのは，若い頃にくらべて，ずいぶん自分がいいかげんになったという感慨です．

良い意味でも悪い意味でもアバウトになってきて，以前は胃が痛くなったような仕事でも，それほど気に病まずにこなせるようになりました．もともと筆者はあまり緻密なアタマをもってはいないと自覚しているのですが，その割に神経質なところがあって，それがこのプログラマという商売にとってはうまく働いているようでした．けれども神経質というのは両刃の刃で，スケジュールの遅れや，自分ではどうしようもない不確定要因に，つい再帰呼び出しのように考えてしまい，すぐスタックオーバフローならぬ，潰瘍を悪くしてしまうのでした．

しかし！　ふと気づくといつの間にやら，考えてもどうしようもないことを考えないようになり，それにともない，潰瘍もいつの間にか，それほど痛まなくなりました．

さらに，年をとると現場の技術にも疎くなります．まだまだ枯れてはいないつもりですが，すべての分野をカバーするがむしゃらな気力はありませんから，若いモノに任せます．ところがそれが暗礁に乗り上げ，連日の残業々々です．サア，以前なら一緒になってコードを追ったりしたものですが，さっぱりわからない分野のこと，「がんばって」と声をかけることくらいしかできません．

ハハア，これだな，と思いました．自分はその分野で役に立たないと割り切っていますから，とても気が楽なのです．いままで自分が苦しんでいるときに声をかけてくれた管理職の上司の気持ちってこうだったんですね．

何にでも首を突っ込んでパタパタしなくなってきたのは，歳を重ねて賢くなったというより，体力がなくなって根気が続かなくなったからなのかもしれませんが，なかなかいいことなのかもしれません．

藤原弘達　(株)JFP　デバイスドライバエンジニア，漫画家

特集

オリジナルアーキテクチャのパソコンを作ろう！

作りながら学ぶコンピュータシステム技術

PC/AT互換機のCPUがまだ386だった頃は，マザーボードにも74TTLがよく使われていた．しかし現在では高速化/高機能化が進み，あらゆる機能がSuperI/Oなどのチップセットに集積されるようになっている．そのためバスやインターフェースの基礎技術を学習/体験しようにも，機能がLSIの中に入ってしまっていて，その動きを見ることが困難になっている．

また，組み込み向けCPUにおいても，SDRAMコントローラやPCIバスコントローラが内蔵され，リファレンスマニュアルどおりにピンを接続してレジスタを初期化するだけで，動作原理を理解しなくても動くシステムができてしまう．これでは，本当の意味でPCIやSDRAMを理解することはできない．

そこで本特集では，コンピュータシステムを構成する各種要素技術を，実際に試しながら実践して学べるよう，FPGAにより各種ハードウェアを実現していく．

ホストCPUにはSH-4を採用し，ローカルバスをFPGAに直結する．メインメモリとしてはSDRAMコントローラを，I/O拡張インターフェースとしてはPCIホストコントローラを設計する．そしてPCIバス上に，グラフィックス表示，キーボード＆マウス入力，ストレージインターフェースを実現する．

プロローグ

実際に動くと感動ものだよ！　おもしろいことやろうぜ！

これがオリジナル仕様コンピュータシステムだ！

原案：井倉将実

1 コトの始まり…

● わかっていないヤツが多いゾ！

某日某所，某居酒屋にて－－－

筆者(以下"筆")：「いやぁ～昨日の客先には参ったよ……」

編集者(以下"編")：「何かあったんですか？」

筆：「某社の組み込みCPUを採用したボードなんだけど，メモリテストで落ちるからハードが悪いのじゃないか！って言われて，現場行ってみたら，相手のソフト屋が『なんか治ったみたいです』だって」

編：「手間がかからなくてよかったじゃないですか」

筆：「それはいいんだけどさぁ～，原因とか聞いたら，SDRAMコントローラの設定パラメータなんだけど，設定値を2から3にしたらメモリチェックプログラムが落ちなくなったので，それで使っています……だって．そのパラメータの変更が，何を意味するのがちゃんと理解してんのか？　おまえ本当にわかってんのか！と小一時間問い詰めたい……」

編：「最近の組み込みCPUのバスコントローラは使いやすくなってるから，よくわからなくても使えちゃうってことですか．デバイスメーカーのアプリケーションマニュアルのとおりにCPUとメモリを接続して，バスコントローラのレジスタを適当にセットすると，PC100[注1]のSDRAMでもなんとなく動いちゃう．けど，SDRAMを本当に理解して使ってるかどうかは，はなはだ怪しい……」

筆：「たしかに，開発期間がどんどん短くなっているので，できあいの設計リソースを使う必要性は認めますよ．でも技術者として，ホントにそれでいいのかなと常々思います」

編：「現状ではなかなか難しいかもしれませんが，技術をきちんと理解して使ってほしいですね」

筆：「難しくないよぉ～．PCIなんて簡単だよ．SDRAMくらい扱えなくてどうする．できない技術者は看板たたんでファミレスでバイトでもしていろって！」

編：「それはファミレスのバイトの方に失礼な．せめて大学戻って出直して来いとか……」

筆：「今の大学って，SDRAMとかちゃんと教えているのかな？」

● より実践的なコンピュータシステム学習教材

編：「SDRAMではないですが，以前Interface誌でCPUの設計記事を連載で掲載したのですが，そのときの先生が『米国の学生はCPUも設計してるぞ！』と，学生にハッパをかけられたのを思い出しますね」

筆：「理系離れが叫ばれている今，日本でCPUを設計できる学生が何人いるかな？」

編：「そうなんですよ．せめてあのぶ厚い"ヘネパタ本"[注2]くらいは読んでほしいけど，はたして読むか？」(**写真1**)

筆：「寝てしまうな……」

編：「なので，アカデミックでハイソ(？)な話は大御所におまかせして，こっちは実践しながらもっと楽しんで学べるような，書籍とか教材とかキットを実現したいなぁ～と思ってるんですよ」

筆：「キットねぇ～」

編：「本当のところをいえばCPUから設計したいんですが……」

筆：「いきなりそれやりますか……」

編：「いや以前，DWM(デザインウェーブマガジン)でCPU設計の特集をやりましたが，いざ使おうとすると，やっぱりソフトウェア開発ツールが問題になると思うんですよ．C++対応とまでいわないまでも，Cコンパイラは欲しい．そしてデバッガ

〔写真1〕ヘネシー＆パターソン コンピュータ・アーキテクチャ
(富田眞治/村上和彰/新實治男訳，日経BP社)

注1：PC100：SDRAMモジュール(DIMM)の規格名称で，クロック100MHz動作対応のもの．
注2：ヘネパタ本：「ヘネシー＆パターソン コンピュータ・アーキテクチャ」の俗称．

も，そこまで用意するのは相当たいへんだと思うんですよ」

筆：「でも，いずれはCPUも設計したいね……」

編：「自分でFPGA[注3]で作ったCPU，キャッシュ，メインメモリコントローラ，PCIバスコントローラでシステムが起動したら，感動ものだと思うんですよぉ〜」

● 要素技術は書いてきたので……

編：「で，ひとまずCPUはおいといて……足まわりといえば，今までいろんな記事を書いていただきましたねぇ〜」

筆：「いっぱい書かされた……」

編：〔されたって……(汗)〕「PCIターゲットから始まって，PowerPCにSDRAMつないだり，PCIバス上にDIMMを接続して大容量RAMカードを作ったり……」

筆：「SH-4のPCIブリッジやIDEインターフェース，PCカードインターフェースも作ったなぁ〜」

編：「グラフィックスカードは，ハードそのものの解説記事はまだ載せてなかったですね」

筆：「そうね，まだ書いてないねぇ〜」

編：「……ふと思ったんですが，いままでのノウハウを集大成して，キーボードまで付けたら，パソコン作れますよね……」

筆：(“キラ〜ン”と目が光る！)「作る？　でも，その昔によく載ってた，74TTL[注4]とせいぜいGAL[注5]を並べたくらいの，マイコンに毛が生えた程度のシステムじゃないんでしょ？」

編：「もちろん．今のパソコンと比較しても遜色ない，現役でも使われているバスやインターフェースをそのまま採用したいんですよ！」

筆：「SDRAMとかPCIとか？」

編：「そうです．ローカルバスはクロック100MHz以上でバス幅64ビット，PCIバス上に各種インターフェース，ストレージはIDE，ハイレゾ[注6]も映る画面にキーボードとマウス！」

筆：「それはたしかにパソコンだな……」

● 使える部品は使ってしまえ

編：「ただし現実問題，たとえばHDDを一から作るか…というとそれは無理なんで，HDDとかキーボードとか，そういうのは入手性の良い市販品/標準品を採用すればいいと思うんですよ」

筆：「つまり秋葉で棄てるほど売られている，メモリモジュールとかHDDとかCD-ROMドライブとかキーボードとか筐体とか，使える部品はそのまま使って，マザーボード部分を作るってこと？」

編：「そうです！　で，やるならATアーキテクチャみたいな酷い(!?)ものはダメですよ．となると必然的にSuperI/Oチップセットとかは使用禁止！」

筆：「じゃ，チップセットをFPGAで作るの？」

編：「そうです！　チップセットはすべてFPGAで実現！　HDLソースは全公開，Linuxがオープンソースなら，こっちはオープンハード！」

筆：「HDLソース公開っすか……．また商売にならない話を……(涙)」

編：「いや，ですから，勉強用に理解しやすいように，もっとも基本的なモードとかタイミングにだけ対応した，簡単なやつでいいんですよ．パフォーマンスが悪くてもOK！　パフォーマンスを追求した高性能設計は，御社の商品として……」

筆：「なんかいいように丸め込まれている気がするナ……」

2 CPUの選定から仕様決定まで

● CPU

編：「さて，CPUですが，どれを使いましょ……」

筆：(間髪入れずに)「68000シリーズ！」

編：(すかさず速攻で)「却下！」

筆：「じゃ，PowerPC G4」

編：「お勉強用という側面もあるんですから，いきなりそんなハイパフォーマンスな石でなくて……」

筆：「じゃ，しょうがなくSH-4」

編：「SH-4，嫌いなんですか？(汗)」

筆：「64ビットデータバスをもっていて，安価で入手性がいいデバイス……となると，SH-4くらいなんですよねぇ〜」

編：「PowerPC系やMIPS系でも，かなり上のクラスでないと64ビットバスもってないですからね」

筆：「組み込み向けとなった時点で，最大バス幅が32ビットになってしまうのが多いんですよね．データバスで使うよりは周辺機能のためにI/Oピンを使いたいってことですかね」

編：「そうですね．今回はプロセッサ的に使うので，周辺機能が多いともったいない……」

筆：「そうそう．A-D/D-Aくらいならまだ許すけど，LCDコントローラとか，Ethrnetコントローラとかを内蔵されると，それを使わなかったらそのCPUを採用するが意味ない……という話も出てくるし」

編：「そう考えるとSH-4，とくにSH7750って，用途の決まった専用周辺コントローラが何にもないんですよね……シリアルコントローラだって，クロック同期と調歩同期の一般的なやつだし……」

筆：「データバス幅64ビットで使うとポート機能が使えないしね！」

編：「そうそう，タイマ/カウンタなんてホント，時間とクロックをカウントするしか芸がない(^^;)」

筆：「……なんだかSH-4の悪口ばっかり言ってますけど，私は

注3：FPGA：Field Programmable Gate Array.
注4：74TTL：74LS00や74HC245などのロジックIC.
注5：GAL：Generic Array Logic.
注6：ハイレゾ：ハイレゾリューションの略．ここではXGA(1024×768ドット)程度の解像度を示す.

〔写真2〕SH7750R と Spartan-II

SH-4は悪くないと思いますよ」(一応フォロー？)
編：「じゃメインCPUはSH7750で決まりぃ〜」
筆：「R版[注7]がすぐ手に入ればいいけどねぇ〜」(**写真2**左)

● FPGA

編：「CPUローカルバスコントローラと，64ビットSDRAMコントローラ，あとPCIホストコントローラだと，ゲート数はどのくらい要りそうでしょうか？」
筆：「15万ゲートくらいあれば問題ないですよ．外部バスが100MHz程度なら，X社のスパII[注8]かな．安いし(笑)」(**写真2**右)
編：「パッケージやピン数は……」
筆：「……これ(ローカルバスとメモリとPCIホスト)を1チップにするんでしょ？　456FG[注9]でもピン数ギリギリじゃないのかな……」

● メモリ

編：「メモリはSDRAMですね．DDR-SDRAMとかいきますか？」
筆：「まさか．それにクロックが100MHz程度なら，DDRよりただのSDRAMのほうがパフォーマンスがいいです．DDRは266とか333くらいのクロックでないと意味ないし……」
編：「で，メモリはモジュール使ってくださいよ．DIMMでもSO-DIMMでもいいですが……」
筆：「ここまで(CPUとメモリとPCIホスト)を1枚のハーフサイズのPCIボードに実装するなら，SOだな……」
編：「SDRAMコントローラはFPGAで設計してもらわないと，SDRAMの勉強になりませんからね」
筆：「それに，SDRAMコントローラとしてSH7750内蔵のを使ってしまったら，外部バスマスタがメインメモリにアクセスできなくなるんで，SH7750を使ってシステムを組むなら外付けのSDRAMコントローラは必須だね」
編：「Rバージョン使ってもSH-4は外部バスが120MHzまでなんで，PC133とはいいませんが，PC100は……」
筆：「理解しやすいようにHDLソースを書くんなら，相当高速なFPGAでないと…….スパIIでチューンしないでPC100はきついよ」

● PCI

編：「システムの拡張性を考えれば，いずれかの業界標準バスは必須でしょう．で，現状で手ごろな標準拡張バスとなれば，PCIバスになるかと思うんですが．たしかにPC/AT互換機から誕生したバスですが，バス自体は汎用性があるし……」
筆：「割り込みまわりとか，ちょっと気に入らないけどねぇ〜漢(おとこ)はやっぱり“ベクタ割り込み”でしょ！(笑)」
編：「たしかに，割り込みルーチンがステータスをチェックしながらバトンタッチして割り込みラインを共有するってのは，パフォーマンス的にいただけないですが…….まぁ〜ATアーキテクチャの上で拡張可能で，かつIRQ共有可能な割り込みシステムということで，いまのPCIバスの割り込み機構に落ちついたんでしょうけど」
筆：「あとね，性善説(？)に基づいたバスアービトレーションの思想も気に入らない．一人でバスを占有し続けるデバイスがいたら，問答無用でレッドカード渡すようなアービタでないと．VME万歳！(笑)」
編：「すいません．VMEバスは勉強不足で……」
筆：「まぁ〜何にせよ，32ビット/33MHzの一般的なPCIバスがいいんじゃないですか？　クロックは66MHzまではいけると思いますが，バス幅は……ホストコントローラに使うFPGAのピン数しだいだな……」
編：「ローカルのメモリはCPUの近いところに置くべきですが，それ以外はすべてPCIバス上に配置して問題ないと思います」
筆：「ホントはグラフィックスくらいはローカルバス直下に置きたいんだけどねぇ〜」
編：「それはまぁ〜，今後の展開への課題ということで……」

● グラフィックス

編：「で，そのグラフィックスですが……」
筆：「PCIバススロットに，そこらのAT用のVGAカードを差し込むのはダメなんでしょう？」
編：「そうですね．アーキテクチャのすっきりしたグラフィックスボードを設計しましょう」
筆：「まさかテキストプレーンとグラフィックプレーンの重ね合わせぇ〜とか，スプライトぉ〜だとか，はたまたポリゴンだのテクスチャマッピングだのいわないよね？」
編：「それのソースまで出していただけるなら嬉しいですが……」
筆：「勘弁してよぉ〜(涙)」
編：「いや，わかってます(汗)．VGAとかXGAで，フルカラーモード固定でいいですよ．アクセラレーション機能もいりませ

注7：R版：SH7750R．従来のSH7750/7750Sと比較し，キャッシュ容量が倍になりキャッシュアルゴリズムが改良された，最高動作クロック240MHzのSH-4．
注8：スパII：ザイリンクス社のFPGA Spartan-II．
注9：456FG：Spartan-IIのパッケージを示す略号．456ピンファインピッチBGA．

ん．とりあえずプレーンなフレームバッファで」

● **キーボード＆マウス**

編：「画面出力ときたら，次は入力デバイスですね．やはりこれから設計するシステムなら，USB インターフェースを採用して USB キーボードと USB マウスなんですかね……」

筆：「あのさ，……キーボードパワー ON とか，させたくない？」

編：（マニアックな機能だ ^^;）「採用！　……となると，USB キーボードだと難しいですね．PS/2 キーボードを常時動かしますか……」

筆：「キーボードコントローラが必要かなぁ」

編：「“AMIKEY”とか，AT 用のコントローラ採用するのは却下ですよ！」

筆：「じゃぁ～ H8 か何かマイコン使って，PS/2 デバイスの汚い（？）ところを綺麗に整形して，PCI バス経由でホストに渡すアーキテクチャにしましょう」

● **ストレージ**

編：「ストレージはもう IDE ですね．……いまさら SCSI ですか？」

筆：「ドライブの価格考えたら，IDE しかないっしょ．しっかし，100G バイトが 2 万円でおつりがくるって，絶対どこかおかしいよな……」

編：「リムーバブルメディアとしては，CF カードを採用したいですね」

筆：「TrueIDE モード[注10]で IDE インターフェースにつなげばいいじゃん」

編：「いや，そうすると，活線挿抜非対応になるので……．FDD を採用しない分，活線挿抜可能なリムーバブル媒体として何か考えておかないと……」

筆：「『FD 読みてぇ～』って人には？」

編：「ATAPI 接続のスーパーディスクドライブ[注11]使ってください．あれ，2DD も 2HD も読めますんで」

● **ネットワーク**

編：「Ethernet は，物理層を FPGA で作るってわけにはいかないですよね……」

筆：「そこは外付け φ チップ買ってくるしかないねぇ～．MAC アドレスの問題もあるし．だいたいさ，秋葉歩いてると，PCI の 100Base-T のカードが 1,980 円とかで売られてるんですよね……もう勝負にならないよ……．これは勘弁してぇ～」

編：「お勉強用として，10M で論理層以降でいいんで，いずれは解説記事載せたいですね」

筆：「……それ，誰が作んの？」

編：（ジーっと筆者を見る目……）（笑）

● **その他**

編：「キーボードパワー ON がありなら，指定した時間になったら自動的に電源の入るタイマ起動も欲しいっすね！」

筆：「お，わかってきましたね！（笑）」

編：「バッテリバックアップ付き RTC で，タイマ一致出力付き……なんて都合のいい石はありませんか？」

筆：「そんなの見たことないなぁ～．あっても高かったり，入手性悪そう……」

編：「であれば，普通の RTC を使って……．キーボード変換用のマイコンでタイマ時間をチェックさせて，それで ATX 電源を制御させましょう！」

筆：「ついでにキーボードでテレビコントロールやらなんやらもほしいんですが……」

編：「もしや，某パソコンをめざしているんじゃ……」

筆：「ますますマニアックな仕様になってきた！（笑）」

● **そして詳細仕様……**

……喧喧諤諤……

筆：「……じゃ，システムクロックは PLL 使って 100MHz をそれぞれに分配しますか……」

編：「いや，SH-4 のシステムクロックって，40MHz とか 30MHz とかまでしか入力できないんですよ．なので，クロックはここから SH-4 にぶちこんで，SH で逓倍したクロックを CKIO から出力させて，それに同期して FPGA を動かすしか……」

……喧喧諤諤……

編：「できればメモリは，SPD[注12]を読んでバンク数や CL 値をプログラマブルにしたいですね……」

筆：「いやぁ～そうすると，SDRAM コントローラの出力段が MPX のおばけになって，高クロックで動かなくなるよ……」

……喧喧諤諤……

筆：「456 ピン BGA といっても，ユーザー I/O ピンは 300 本くらいですからね．それじゃ信号線足りませんよ……」

編：「そうか……，じゃあ MPX モード時はアドレスバスは使わないんで，ここをこうして……」

筆：「うぅ～わ，鬼ぃ～．そこまでして詰め込みますか……．I/O ピン使用率 100%だなんて……」

……喧喧諤諤……

3 これがオリジナル仕様コンピュータシステムだ！

ということで，最終的にまとまったオリジナル仕様コンピュータシステムの外観を**写真 3** に，ブロック図を**図 1** に，各仕様を**表 1** に示します．

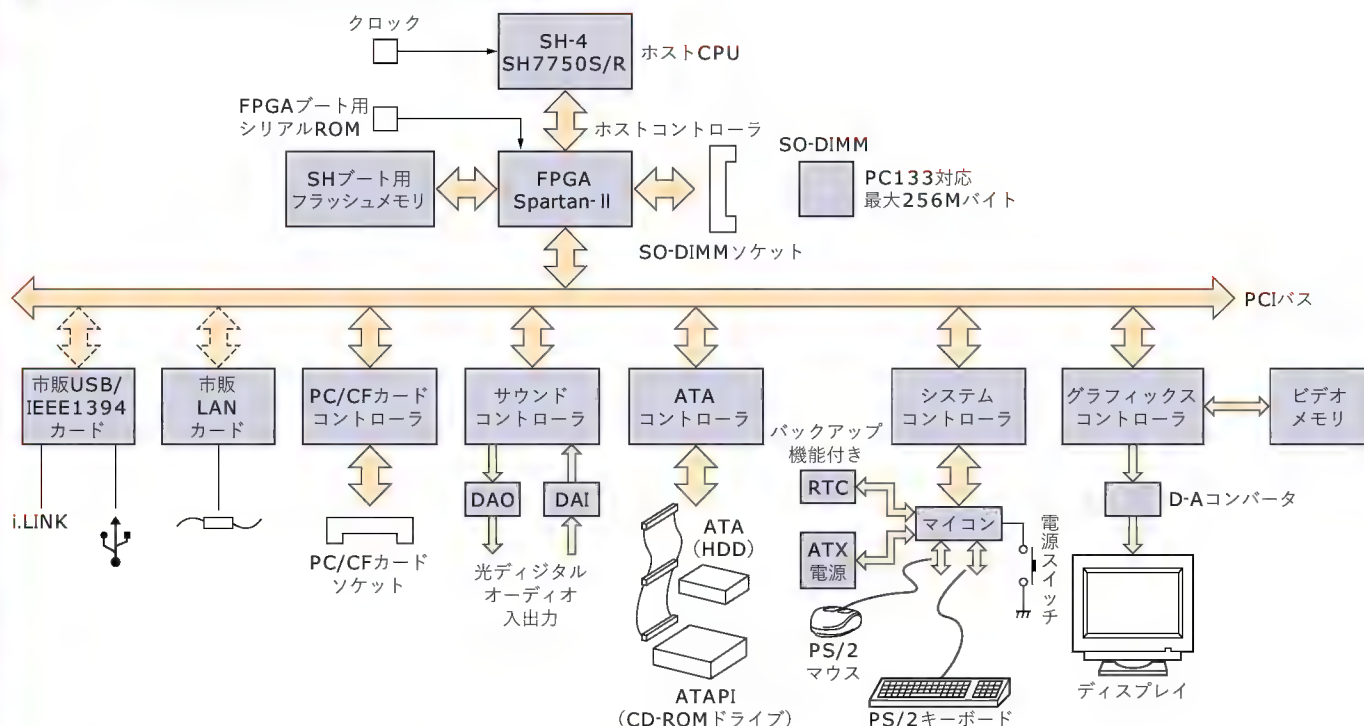
● **最大 240MHz 駆動のホスト CPU SH-4**

ホスト CPU は，最大 64 ビットのバス幅に対応している SH-4 SH7750S または R（日立製作所）を採用しています．R バージョンはそれまでの SH-4 と比較し，キャッシュ容量が倍になり，ま

注 10：TrueIDE モード：CompactFlash カードを IDE として使うモード．

注 11：スーパーディスクドライブ：容量 120M/240M バイトの大容量フロッピーディスクドライブ．3.5 インチ 2HD/2DD のフロッピーディスクも読み込める．

注 12：SPD：Serial Presence Detect．第 2 章参照のこと

〔図1〕オリジナル仕様コンピュータシステムのブロック図

〔写真3〕オリジナル仕様コンピュータシステムの外観
(液晶ディスプレイやキーボード，HDD，マイクロATX筐体は市販のものを流用．BMPローダでBMPファイルを表示したところ)

たキャッシュアルゴリズムも強化されているので，同じクロック数のSバージョンなどよりパフォーマンスの向上が期待できます．

SH7750Sでは最大200MHzクロック動作，SH7750Rでは最大240MHzクロック動作に対応していますが，RバージョンのSH-4は，まだ潤沢に入手できる状態ではないようで，筆者のところでも，200MHz品のSH7750Rを，サンプルとして2個，なんとか入手できただけです．本誌が発売される頃には240MHz品が入手しやすくなるでしょうか．

本特集では入手しやすいSH7750Sを，外部バスクロック66MHzで動作させています．

● **最大256Mバイトのメインメモリ SDRAM**

メインメモリとしてはPC133対応のSDRAMを搭載したSO-DIMMソケットを実装しています．64Mバイトから最大256Mバイトまでの容量を搭載可能です．ただし外部バスクロックが，SH-4がRバージョンでも最大120MHzなので，PC133を使う場合でも133MHz駆動はできません．この場合は120MHz駆動となります．

本特集では外部バスクロックを66MHzで動作させているのでPC66対応を，また容量として64Mまたは128MバイトのSO-DIMMの使用を想定した設計を解説します．

● **SH-4ブート用フラッシュメモリ**

SH-4の各種初期化プログラムやOSのローダなど，IPLを格納するROMとして容量2Mバイトのフラッシュメモリを搭載しています．

● **ホストコントローラ FPGA Spartan-II**

本システムの要ともなるホストコントローラは，456ピンBGAのFPGA Spartan-II(ザイリンクス)を採用しています．これにより，SH-4ローカルバス，メインメモリ，ブート用フラッシュROM，そしてPCIバスを制御します．SDRAMコントローラをホストコントローラ内に内蔵するので，PCIバスマスタからのメインメモリへのアクセスにも対応しています．

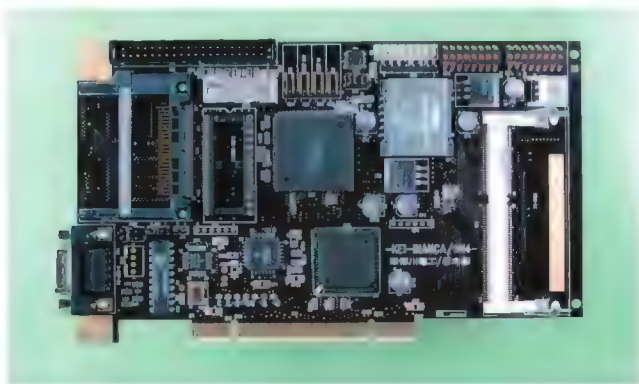
● **システムバスはPCI**

拡張用標準バスとしてもっとも普及しているPCIバスを採用しています．PCIホストコントローラは最大66MHzのクロックにも対応しますが，本特集では一般的に普及しているクロック

〔表1〕オリジナル仕様コンピュータシステムの仕様

項　目	最大/最高仕様	本特集で解説している仕様
ホストCPU	SH7750S/R：CPUコアクロック最大240MHz/ 外部バスクロック最大120MHz	コアクロック200MHz/バスクロック66MHz
メインメモリ	PC133仕様(120MHz駆動)/64M～最大256MバイトSO-DIMM対応	PC66仕様/64Mバイトまたは128Mバイト
PCIバス	バス幅32ビット/クロック最大66MHz 独自拡張PCIベクタ割り込み対応	バス幅32ビット/クロック33MHz
グラフィックス	VAG/SVGA/XGA解像度/32ビットフルカラー対応 60Hzまたは75Hzリフレッシュレート対応 シュリンク15ピン/アナログRGBコネクタ実装 球面スクロール/V-Sync割り込み/フェイドコントロール機能 BitBlt(矩形転送)/2Dアクセラレーション機能	VGA/32ビットフルカラー/ 60Hzリフレッシュレート シュリンク15ピン/アナログRGBコネクタ実装
キーボード	PS/2キーボード(日本語/英語)対応	←
マウス	PS/2マウス(ホイール&5ボタン)対応	←
ストレージ	最大ATA66対応バスマスタATAインターフェース	PIOモード4対応ATAインターフェース
PCカード/CFカード インターフェース	活線挿抜対応PCカード/CFカードインターフェース I/O系カード対応	－
サウンド	32k/44.1k/48kHz対応/光ディジタルオーディオ入出力(PCM)	－
ネットワーク インターフェース	市販PCI拡張ボードを使用	－
USB & IEEE1394	市販PCI拡張ボードを使用	－
その他	バックアップ機能付きリアルタイムクロック&8KバイトRAM リモート/タイマ起動対応システムコントローラ搭載 キーボードパワーON対応	－

〔写真4〕プロセッサボード(SH-4/SO-DIMM/PCI搭載)

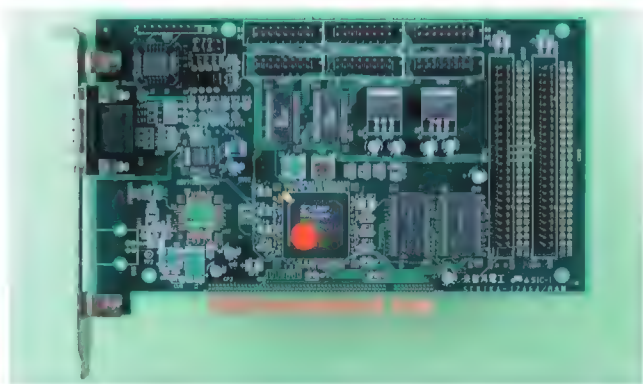
〔写真5〕グラフィックスボード

33MHzに対応したものを解説します．

PCIバスを採用することで，PCIバスに接続可能なさまざまな拡張ボードを接続することができます．

以上，CPU，メインメモリ，ブート用フラッシュメモリ，PCIホストコントローラまでをハーフサイズPCIボードに基板化しました．これをプロセッサボードと呼び，その外観を**写真4**に示します．

以降の各種インターフェースコントローラは，システム的にはすべてPCIバス上にぶら下がる形となり，最終的にはATX形状のマザーボードとして基板化を予定しています．

● グラフィックス

VGA/SVGA/XGAの各解像度に対応した，32ビットフルカラーグラフィックスボードです．テキスト画面モードはないので，コンソールテキストの表示はフォントデータをフレームバッファ上に描画する必要があります．球面スクロール[注13]やV-Sync(垂直同期信号)に同期した割り込みの発生，フェイドコントロール機能やBitBlt(矩形転送)機能など，2Dアクセラレーション機能も内蔵しています．

このうち本特集では，解像度はVGA固定の32ビットフルカラーフレームバッファボード(アクセラレーション機能なし)について解説します(**写真5**)．

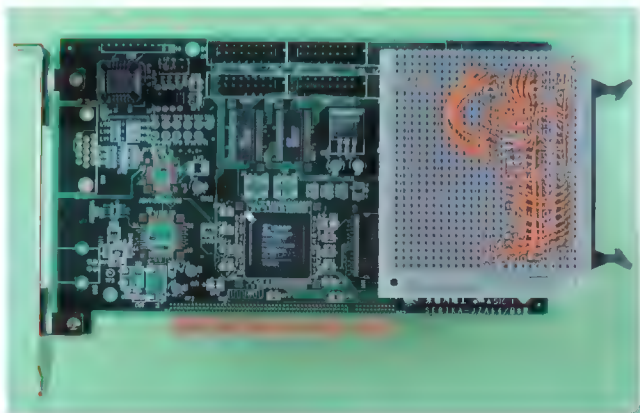
● キーボード＆マウス

PS/2キーボードとPS/2マウスに対応しています．なぜPS/2インターフェースというレガシーインターフェースを採用したかは，すでに説明したように，キーボードパワーON機能を実現したかったからです．

● ATAインターフェース

ストレージとしてもっとも普及し，入手性もよく，価格もこ

注13：球面スクロール：右スクロールなら右端に消えたものが左から表示される，上スクロールなら上端に消えたものが下から表示されるスクロール表示動作．

〔写真6〕試作評価用PCIボードを使って製作したATAインターフェース＆キーボード/マウスインターフェースボード

なれているATAインターフェースを採用することにしました．ただし最新のシリアルATAは，これから普及する規格なので現状では対応デバイスも少ないことから，ATA66対応のバスマスタATAインターフェースを設計/製作しています．

本特集では，もっとも基本的な転送モードであるPIO転送に対応したATAインターフェースについて解説します．

写真6は，ATAインターフェースとキーボード/マウスインターフェースのPCIデバイス部分を1枚のボードにしたものです．よって，このPCIデバイスはマルチファンクションデバイスとなります．

● PCカード/CFカードインターフェース

今回の特集では誌面の都合で解説していませんが，16ビットPCカードやコンパクトフラッシュカードを読み書きするためのインターフェースと，PCIバスに接続するブリッジコントローラを設計/製作しています．

これらのカードは，TrueIDEモードでの接続ではなく，PCカードとしてシステムにマッピングしているので，活線挿抜に対応しています．よってフラッシュATAカードや，ディジカメ用小型フラッシュメモリカードをPCカードやCFカードに変換するアダプタを使用して，このシステムで使用することも可能です．さらに無線LANカードや移動体通信カードなど，I/Oカードを使ってインターフェースを拡張することも可能です．

● サウンドカード

これも今回の特集では誌面の都合で解説していませんが，光ディジタル対応のPCMサウンドカードも設計/製作しています．Windows環境などで作成したWAVファイルを再生して，光ディジタル出力端子から出力し，外付けの光ディジタル対応AVアンプでアナログに変換してスピーカを鳴らしたり，CDプレーヤなどの光ディジタル出力を接続して，ディジタルオーディオ信号を録音し，WAVファイルのフォーマットでファイルとして書き出すプログラムも動作しています．

● ネットワークインターフェース

現在のコンピュータシステムにおいて，もう一つ重要なインターフェースといえばネットワークインターフェース，すなわちEthernetでしょう．

残念ながら筆者はまだEthernetに関するノウハウをもっていないので，ネットワークインターフェースの設計/製作は実現できていません．プロトタイプシステムでも，PCI拡張スロットにPCIバス版NE2000互換の市販のネットワークカードを差し込んで動作させています．NE2000互換なので10Mbpsにしか対応していませんが，ドライバを用意すれば100MbpsのFast Ethernetも動作させることができます．

Ethernetの場合，物理層はそのままFPGAなどで実現することはできませんが，外付けに市販の物理層チップを接続し，論理層以降をFPGAで実現することは可能です．

● USB & IEEE1394

これも，システムバスがPCIバスなので，PCI拡張スロットにホストインターフェースボードを実装することが可能です．最近ではUSB1.1/2.0とIEEE1394を，1枚のPCIボードで拡張可能なボードも市販されています．

これらシリアルバスも，そのままFPGAで直接コントロールすることは難しいのですが，Ethernet同様，物理層を何とかすれば，論理層はFPGAで実現することは容易です．

● その他の機能

バッテリバックアップ機能付きの，8KバイトRAM内蔵リアルタイムクロックを搭載し，指定した時間にシステムを起動させることが可能です．また外部からのリモート信号，またはキーボードの特定のキーを押すことでシステムを起動させることも可能です．

● FPGAはVHDLで設計

今回は，基板化したプロセッサボードにも，試作で使用したPCI評価ボードにも，どちらも同じザイリンクス社のFPGAを採用しています．ザイリンクス社のWebから無償でダウンロードできるWebPACK ISE以外，FPGA開発に必要なツールは入りません．特集で解説したソースはすべて，この無償版のツールでも論理合成/配置配線可能です．今回，HDLにはVHDLを採用しました．

● プロセッサボードなどの入手方法

今回基板を起こしたプロセッサボードや，PCI評価ボードなど各基板の入手方法はエピローグ(p.126)を参照してください．本誌が発売される頃には，各基板の回路図やドキュメントなども整理し，学習キットとして使える体裁に整えたいと思います．もちろん設計したコントローラやファームウェアなどはソースを添付します．

● OS/ソフトウェアについて

現在のところ，まだOSと呼べるほどのものは動作していません．ITRONやLinuxといったOSの移植/開発をしていただける方，大募集中です．

いくら・まさみ　来栖川電工有限会社

第1章

64ビット/MPXバスモード/クロック66MHzで動作させる

SH-4用ローカルバスコントローラの設計/製作

井倉将実

64ビット/100MHzクラスの外部バスをもち，安価で入手しやすいCPUということで，今回のシステムにはSH-4が採用された．ここではまず，SH-4のアーキテクチャや外部バスの動作を解説し，設計したプロセッサボードのブロック図などを解説する．最後に，SH-4とFPGAを接続するローカルバスコントローラを設計/製作する．

（編集部）

はじめに

現在のコンピュータシステムでも通用するバス/インターフェース技術を，試しながら実践的に学習するという，壮大な（無謀な？）な特集に挑戦することになりました．興味をもって読んでいただけるよう，理論よりまずは目の前で動くもの，おもしろいものを……というスタンスで説明します．

● システム構成考察

図1に，一般的なパソコンと組み込み向けCPUを搭載したシステムの構成の違いを示します．

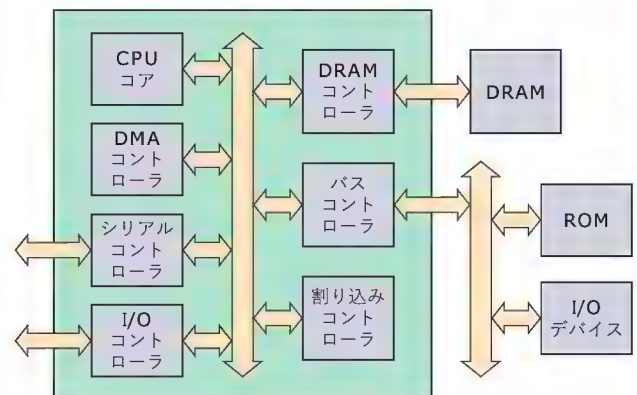
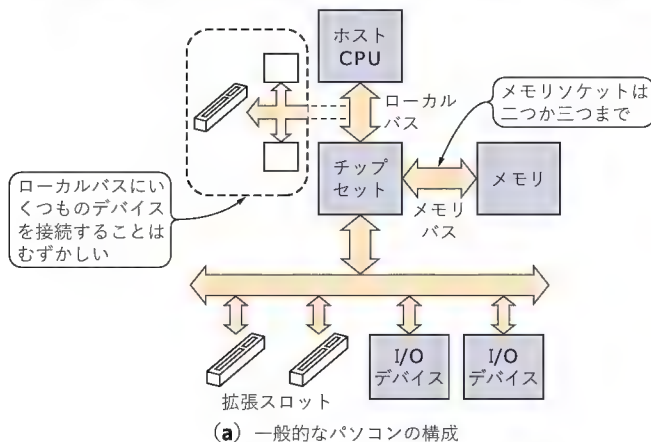
パソコンは，頭にホストCPUがあり，ローカルバス経由でチップセット（ホストコントローラ）が繋がります．チップセットの隣にはメインメモリが，その下には各種I/Oがつながる構造になっています．システムのパフォーマンスはCPUの性能もさることながら，メインメモリの性能も大きく関わってきます．よってローカルバスやメモリバスは，そのシステムの中でいちばん高速なバスとなります．高速に動作させなければならないので，ローカルバスにたくさんのデバイスを接続したり，拡張性を考えてコネクタを配置するということも難しくなっています．逆に各種I/Oが接続されるバスは，性能もさることながら拡張性が重視されます．よってローカルバスやメモリバスより低速ですが，それらのバスよりは数多くのデバイスを接続することが可能です．また，拡張も容易なように拡張スロットも設けることもできます．

各種I/Oが接続されるバスとしては，現在ではPCIバスが一般的です．PCIではデータ転送の効率を上げるために，CPUによるプログラム転送以外に，バスマスタ転送と呼ばれる，いわゆるDMA転送によりバスの帯域を生かしたデータ転送を行います．詳しくは第3章で説明しますが，バスマスタ転送ではPCIバス上からメインメモリへの転送が行われます．つまりメインメモリは，ホストCPU以外にPCIバス側からのアクセスにも対応しなければなりません．そのためチップセットがその間を取り持っているのです．

● シングルホストの組み込み機器

組み込み向けCPUを搭載したシステムでは，CPU内蔵のDRAMコントローラにDRAMを直結し，ローカルバスにROM

〔図1〕一般的なパソコンと組み込み向けCPUを搭載したシステム構成

（a）一般的なパソコンの構成

（b）組み込み向けCPU搭載システム

やSRAM，周辺デバイスを接続しています．さらにCPUに内蔵された機能を使って，各種インターフェースを実現する場合もあります．

このようなシステムでは，ROMやSRAMを接続したバスをローカルバスとも呼べなくはないのですが，本当の意味でのローカルバスではありません．

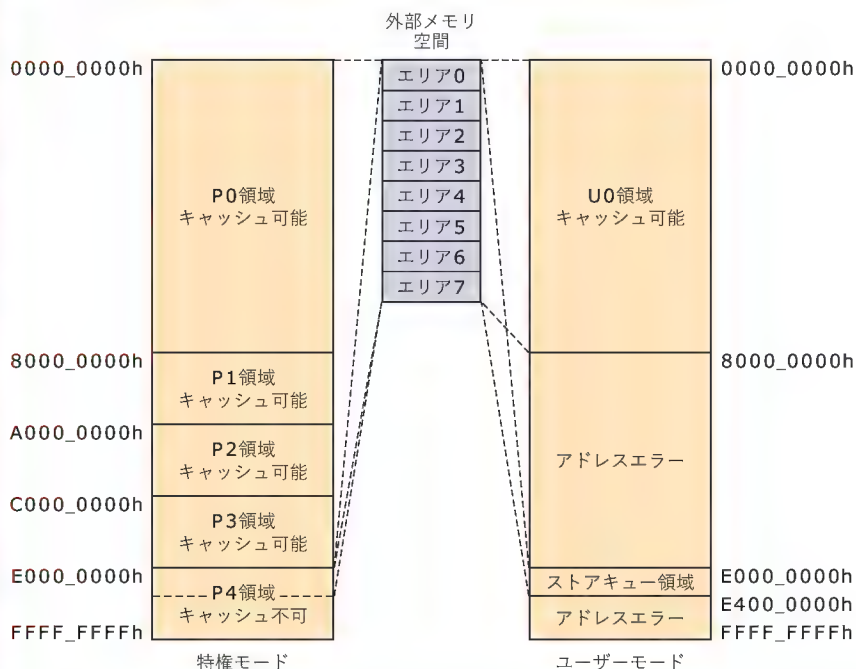
また，DRAMの制御をCPUが握っているため，周辺デバイスがバスマスタとなってメインメモリにデータを転送することはできません．そのような場合にはCPU内蔵のDMAコントローラを使ってデータ転送を行います．メインメモリ側から見ると，アクセスは必ずCPUから行われるわけです．DMAコントローラを使った転送でも，DMAコントローラ自体はCPUに内蔵されているので，アクセスはやはりCPUからに見えます．

● SH-4選択の根拠

本特集ではコンピュータシステムを構成する技術を学習するという目的があるので，**図1(b)**のような組み込みCPU直結システムでは勉強になりません．ローカルバスとメモリバス，そしてI/Oバスをそれぞれ自分で扱ってこそ，目的が達せられます．そうなると，必然的に**図1(a)**の構成を採ることになるでしょう．

さて，そういう意味では，ホストCPUはそれこそPentiumでもよかったのですが，最近のPentiumで使用しているバスは特許などのからみがあるので，自由に使うわけにはいきません．

また，メモリモジュールには市販のものを使うので，DIMMであれSO-DIMMであれ，データバス幅は64ビットとなります．UMAアーキテクチャなど特別な場合を除き，ローカルバスよりメモリバスが広いのは，あまりバランスの良い設計とはいえません．

〔図2〕SH-4のアドレス空間

プロセッサとして使ってもムダがなく(内蔵周辺機能は極力使わない)，データバス幅64ビットで，特許の心配のないごく一般的なローカルバスをもった，安価でしかも入手性の良いデバイス……ということで，今回はSH-4(SH7750)に白羽の矢が立ちました．

1 SH-4の概要

今回のシステムはホストCPUとしてSH-4(SH7750)を採用しました．今回の目的には最適なCPUだと思います．

誌面の都合で，SH-4についての詳しい解説はできません．詳細は参考文献1)や2)などを参照してください．ここではSH-4用ローカルバスコントローラを設計するうえで必要な，外部バスの動作について詳しく解説します．

1.1 SH-4の外部バスの動作

● P0～P4領域

図2にSH-4のアドレス空間を示します．SH-4は32ビットCPUでアドレス空間も4Gバイトあります．しかし，それがそのまま外部に出力されているわけではありません．本格的なOSを載せた場合のメモリ保護機能のために，特権モードでしかアクセスできない領域や，MMUによるアドレス変換を行う領域/行わない領域，キャッシュ領域/非キャッシュ領域を，32ビットアドレスの上位ビットで切り分け，それぞれをP0～P4領域と呼んでいます．

● エリア0～6までの七つのチップセレクト

さらにSH-4では，外部にメモリや周辺I/Oデバイスを接続しやすいように，エリア0($\overline{\text{CS}}$[0])～エリア6($\overline{\text{CS}}$[6])まで七つのチップセレクトをもっています．たとえば，P1～P4領域は512バイトづつありますが，それぞれの領域からエリア0～エリア6までアクセスできます．これは，たとえばエリア2にメインメモリとしてRAMを置いた場合，これはP0やP1領域のキャッシュ空間からアクセスしたほうがプログラムの実行速度は上がりますが，たとえばエリア4にI/Oデバイスを接続した場合は，I/Oアクセスにキャッシュが効いてしまってはまずいので，P2領域からアクセスします．

図2ではエリアが7まであるのに，チップセレクトとしてはエリア6までしかないのは，エリア7にCPU内蔵の周辺機能のレジスタなどを配置するためです．SH-4にはI/O空間がないので，すべてメモリマップドI/Oとなります．そのため$\overline{\text{CS}}$[7]という信号は外部に出ていません．

一つのエリアは64Mバイトのサイズがあ

〔表1〕
各エリアに接続可能なメモリやバス幅

エリア	外部アドレス	容量(バイト)	接続可能メモリ	設定可能バス幅	アクセスサイズ
0	0000-0000h ～ 03FF-FFFFh	64M	SRAM	8, 16, 32, 64[*1]	8, 16, 32, 64[*6] ビット 32バイト
			バースト ROM	8, 16, 32[*1], 64[*7]	
			MPX	32, 64[*1]	
1	0400-0000h ～ 07FF-FFFFh	64M	SRAM	8, 16, 32, 64[*2]	8, 16, 32, 64[*6] ビット 32バイト
			MPX	32, 64[*2]	
			バイト制御 SRAM	16, 32, 64[*2]	
2	0800-0000h ～ 0BFF-FFFFh	64M	SRAM	8, 16, 32, 64[*2]	8, 16, 32, 64[*6] ビット 32バイト
			SDRAM	32, 64[*2 *3]	
			DRAM	16, 32[*2 *3]	
			MPX	32, 64[*2]	
3	0C00-0000h ～ 0FFF-FFFFh	64M	SRAM	8, 16, 32, 64[*2]	8, 16, 32, 64[*6] ビット 32バイト
			SDRAM	32, 64[*2 *3]	
			DRAM	16, 32, 64[*2 *3]	
			MPX	32, 64[*2]	
4	1000-0000h ～ 13FF-FFFFh	64M	SRAM	8, 16, 32, 64[*2]	8, 16, 32, 64[*6] ビット 32バイト
			MPX	32, 64[*2]	
			バイト制御 SRAM	16, 32, 64[*2]	
5	1400-0000h ～ 17FF-FFFFh	64M	SRAM	8, 16, 32, 64[*2]	8, 16, 32, 64[*6] ビット 32バイト
			MPX	32, 64[*2]	
			バースト ROM	8, 16, 32[*2], 64[*7]	
			PCMCIA	8, 16[*2 *4]	
6	1800-0000h ～ 1BFF-FFFFh	64M	SRAM	8, 16, 32, 64[*2]	8, 16, 32, 64[*6] ビット 32バイト
			MPX	32, 64[*2]	
			バースト ROM	8, 16, 32[*2], 64[*7]	
			PCMCIA	8, 16[*2 *4]	

注 ＊1：外部ピンでメモリバス幅を指定
＊2：レジスタでメモリバス幅を指定
＊3：SDRAM インターフェース時は，バス幅は 32，64 ビットのみ
また，DRAM インターフェース時は，バス幅はエリア 2 では 16，32 ビットのみ，エリア 3 では 16，32，64 ビットのみ
＊4：PCMCIA インターフェース時は，バス幅は 8，16 ビットのいずれかのみ
＊5：予約エリアはアクセスしないでください．アクセスした場合は動作の保証はできない
＊6：アクセスサイズが 64 ビットとなるのは DMAC による転送(CHCRn.TS=000)の場合のみ
FMOV(FPSCR.SZ=1)による外部メモリへのアクセスの場合，アクセスサイズが 32 ビットの転送が 2 回行われる
＊7：SH7750R のみ設定可

るので，外部メモリは最大で 448M バイトまで接続可能となります．

● **豊富なバスモード**

SH-4 の外部バスは，さまざまな種類のメモリや周辺 I/O デバイスを，グルーロジックなしで接続できるように，いろいろなバスモードを備えています．**表 1** に各エリアに接続可能なメモリやバス幅などを示します．

接続可能なデバイスとして“SRAM”と書かれているのは，アドレスバス/データバスとメモリリード/ライト信号による，ごく一般的なバスアクセスを示します．ここでは標準バスモードと呼ぶことにします．標準バスモードは，すべてのエリアで使用可能です．

また特定のエリアにしか接続できないデバイスもあります．たとえば，SDRAM はエリア 2 と 3，バースト ROM はエリア 1，5，6，PCMCIA 機能はエリア 5，6 にしか接続できません．

標準バスモードと並んですべてのエリアで使用できるモードに MPX バスモードがあります．MPX とはその名のとおり，アドレスバスとデータバスがマルチプレクスされたモードです．

それではもっともよく使われる標準バスモードと，今回のシステムでも採用した MPX バスモードについて，もう少し詳しく説明します．

1.2 標準バスインターフェース

● **標準バスモードで使う信号**

アドレスバスとデータバスを使ったごく一般的なバスの動作のモードです．もっともバスアクセス動作を理解しやすいモードだと思います．使用される SH-4 の各信号を示します．

- CKIO ：バスクロック
- $\overline{BS}$ ：バスサイクルスタート
- $\overline{CS}$[6：0] ：チップセレクト
- A[25：00] ：アドレスバス
- D[63：00] ：データバス
- $\overline{WE}$[7：0] ：ライトイネーブル
- $\overline{RD}$ ：リード
- RD/$\overline{WR}$ ：リード(正論理)/ライト(負論理)
- $\overline{RDY}$ ：レディ

● 基本的なバスの動作

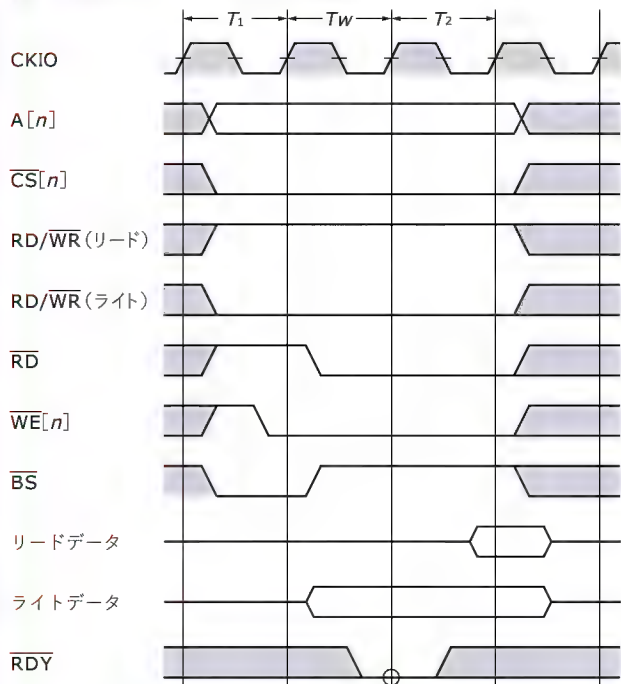
図3 に SH-4 の標準バスモードでのバスの動作波形を示します．

最初のバスクロック(T_1)の立ち上がりでアドレスバスと $\overline{CS}$[n]，RD/$\overline{WR}$信号が出力されます．またバスサイクルの開始を示す $\overline{BS}$ 信号が 1 クロック期間だけアサートされます．

ライトサイクルの場合は，T_1 の立ち下がりのタイミングで $\overline{WR}$ がアサートされ，ライトデータは次のバスクロック(T_2)の立ち上がりのタイミングで出てきます．アドレスが出力された時点で同時にライトデータも出力される CPU もありますが，SH-4 のライトデータは 1 クロック遅れて出力されることになります．ライトデータは T_2 クロックの次のクロックの立ち上がりまで出力されます．

リードサイクルの場合は，2 クロック目の立ちあがりから $\overline{RD}$ 信号が出力されます．CPU が実際にデータを読み込むのは，T_2 クロックの次のクロックの立ち上がりクロックのタイミングです．

SH-4 の基本バスサイクルは最小 2 クロックです．ただし，この場合は外からウェイトを挿入することができない，完全ノーウェイト動作です．

ウェイトを挿入するには，ウェイト数が固定でよければ CPU 内蔵のバスコントローラにウェイト数を設定することで，自動的に T_1 と T_2 の間に T_w を挿入することができます．外からウェイトのタイミングを挿入する場合は，2 クロック目以降のバスクロックの立ち上がりで $\overline{RDY}$ 信号を“H”レベルにしておけば，そのクロックはウェイトクロックとなり，バスの状態はそのまま保持されます．クロックの立ち上がりで $\overline{RDY}$ 信号を“L”レベルにすると，そのクロックがバスアクセスの終了クロックとなる T_2 になります．つまり外部からウェイトを挿入した場合は，$\overline{RDY}$ 信号を“L”レベルにした次のクロックでデータ転送が成立(リードサイクルならデータ読み込み，ライトサイクルならデータ出力終了)することになります．

〔図3〕標準バスモードでのバスの動作

外からウェイトを挿入できる状態で最短のバスアクセスは，T_w が 1 クロックだけ挿入された 3 クロックサイクルとなります．

● 連続アクセスの場合

連続したアドレスに間をあけずにアクセスした場合は，**図3** の T_2 のクロックの次のクロックがそのまま T_1 クロックとなります．このとき，同じエリアにアクセスした場合は，直前のバスアクセスでアサートした $\overline{CS}$[n]信号がアサートされたまま次のバスアクセスが始まります．実はチップセレクトは，SH-4 内部の A[28：26]を単純にデコードしただけの信号なので，同一エリア内の連続アクセスでは $\overline{CS}$[n]信号がアサートされたままとなります．

またリード/ライト方向を示す RD/$\overline{WR}$信号も，リードサイクルなら“H”レベルのまま，ライトサイクルなら“L”レベルのままとなります．

● 低速デバイスを接続する場合

リード信号がネゲートされてから実際にデータバスのドライブが終了するまで時間のかかる低速なデバイスでは，リードサイクルからライトサイクルへ切り替わったとき，データバスがぶつかってしまう場合もあります．SH-4 では異なるエリアに連続してアクセスする場合や，アクセス方向が切り替わる場合，このようなデータバスの衝突を防ぐためにアイドルサイクルを挿入する機能もあります．詳しくはバスコントローラの説明を参照ください．

ちなみに，今回ホストコントローラとして採用する FPGA は十分に高速なので，アイドルサイクルの挿入は不要です．

● リード信号とライトイネーブル信号

SH-4 はバイト(8 ビット)/ワード(16 ビット)/ロングワード(32 ビット)，さらに内蔵 DMA コントローラの設定でクワッドワード(64 ビット)でのバスアクセスが可能です．しかし，リードサイクル時に出力されるのは $\overline{RD}$ 信号だけで，アクセスサイズを示す信号がないことにお気づきでしょうか．

もちろん CPU コアそのものは，各種サイズでのアクセスが実行できますが，そのアクセスがバスコントローラを経由して外部に出力されると，リードサイクル時は，そのエリアで設定されているバス幅で一気に読み出す動作をします．そして必要なバイト位置のデータのみ CPU コアに返すという動作をしているようです．

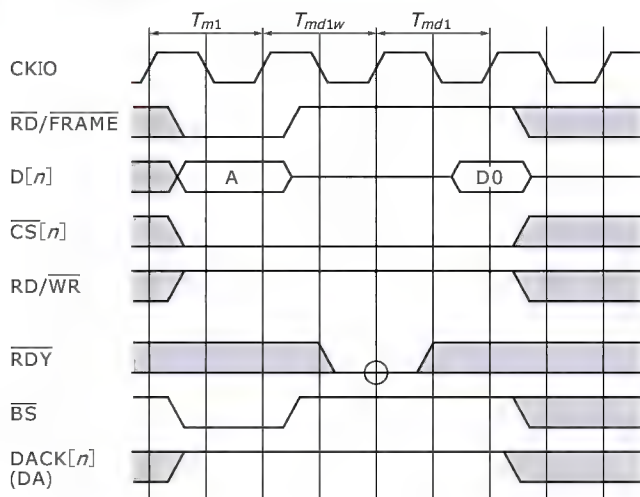
このようなリードサイクル時の動作は，SH シリーズの伝統のようです．

しかし，さすがにライトサイクルでは，バイト単位での書き替えも可能なように，最大 64 ビットバス幅に合わせて，$\overline{WR}$[n]信号が 8 本用意されています．

● バイト制御 SRAM モード

しかし，やはりリードサイクル時もバイトイネーブル情報が欲

〔図4〕MPXバスモード/ノーウェイト/シングルリード時のバスの動作波形

〔図5〕MPXバスモード/ノーウェイト/シングルライト時のバスの動作波形

しいという声に応じたのか，SH-4ではWR[n]信号がリード時にもアサートされる，バイト制御SRAMモードが用意されています．ただしこのモードが使えるのはエリア1とエリア4のみです．

● **バーストROMモードほか**

最初のアドレス設定に数クロックかかるが，以降の連続したアドレスのアクセスにはより短い時間でアクセスできるバーストROMを接続するためのモードや，PCカードを接続するためのPCMCIAモードなど，さまざまなバスのモードがありますが，ここでは説明を省略します．

1.3 MPXバスインターフェース

● **データバスをマルチプレクスで使う**

MPXバスモードは，データバスをアドレスとデータの時分割に利用するモードです．MPXバスモードで使う信号は次のようになります．

- CKIO　：バスクロック
- BS　：バスサイクルスタート
- FRAME　：アドレスフレーム
- CS[6：0]　：エリアチップセレクト
- D[63：00]　：アドレス/データ時分割
- RD/WR　：リード(正論理)/ライト(負論理)
- RDY　：レディ

標準バスモードにはなかったFRAME信号が新たに出てきましたが，これはまったく新しい信号というわけではなく，実体はRD信号そのもので，MPXバスモードのときの信号名称となります．また標準バスモードで使っていたアドレスバスおよびWR[n]信号がなくなっています．これらの信号はMPXバスモードでは使いません．

● **シングルリード**

図4にMPXバスモード，シングルリード時のバスの動作波形を示します．

最初のT_{m1}クロックの立ち上がりでFRAME信号はアサートされ，データバスにアドレスが出力されます．またチップセレクトCS[n]信号は標準バスモードと同様にそれぞれアクセスしたエリアごとにアサートされます．さらにアクセス方向を示すRD/WR信号も出力されます．バスサイクルスタートを示すBS信号も，1クロック期間のみ出力され，この期間をアドレスフェーズと呼びます．

次のクロックT_{md1w}はウェイトサイクルで，リードサイクルには必ず挿入されます．というのは，MPXバスモードではデータバスを使ってアドレスを出力するので，最初のクロックではCPUがデータバスをドライブしています．その直後のクロックで選択されたデバイスがデータバスをドライブすると，データバスが衝突する場合があります．よって，データバスをドライブするデバイスが切り替わるところでは，必ず1クロックのウェイトが入ります．

そして，クロックT_{md1w}の次のクロックの立ち上がりでRDY信号のチェックが行われ，外部からのウェイト挿入をチェックします．ここで外部からのウェイト挿入がなければ，そのクロックはT_{md1}クロックとなり，さらにその次のクロックでデータの読み込みが行われます．

● **シングルライト**

図5にMPXバスモード/シングルライト時のバスの動作波形を示します．一目見て，ウェイトクロックがないことがわかります．ライト動作はCPUがデータを出力するので，アドレス出力後にそのまま続けてデータバスをドライブできます．よってリードサイクル時には挿入されたウェイトが必要ありません．

ウェイトクロックが挿入されないので，RDY信号のチェックも，1クロック前倒しになります．

RD/WR信号は，ライトサイクルなので"L"レベルになります．

● **アクセスサイズ情報**

MPXバスモード時のアドレス情報のフォーマットを**図6**に示

〔図6〕MPXバスモード時のアドレス情報のフォーマット

D63/D31	D62/D30	D61/D29	アクセスサイズ
0	0	0	バイト
		1	ワード
	1	0	ロングワード
		1	クワッドワード
1	×	×	32バイトバースト

〔図7〕
MPXバスモード/ノーウェイト/バーストリード時のバスの動作波形

します．標準バスモードでのA[25：0]の情報は，D[25：0]に出力されます．そしてアクセスサイズ情報を示す情報がD[63：61]に出力されます．なお，このアクセスサイズ情報はSH7750RではD[31：29]に出力するようにも設定可能になり，32ビットMPXバスモード時ではデータバス32ビットの配線だけで，アクセスサイズを判定することが可能になりました．

このアクセスサイズ情報で，注目すべきは32バイトバースト転送という項目です．これはSH-4のキャッシュ有効時に，キャッシャブル空間からメモリをアクセスし，キャッシュ1ライン分のデータをバーストアクセスにより読み込んだり，キャッシュをライトバックするために書き込むなどといった用途に使われます．

● バーストリード

図7にMPXバスモード/バーストリード時のバスの動作波形を示します．SH-4のキャッシュは1ライン32バイトです．よってバーストアクセス時は，合計32バイト分を一気に転送します．アクセス先のエリアのバス幅が64ビットなら4ワードの連続アクセスを，32ビットなら8ワードの連続アクセスが発生するわけです．

先ほどのシングルアクセスでは，FRAME信号はBS信号と同じで1クロック期間しかアサートされませんでしたが，バーストアクセスの場合は，連続転送の要求がある間はFRAME信号は**図4**のようにアサートされ続けます．最後の一つ前のデータ転送が成立した時点でネゲートされ，最後のワードはシングルアクセスと同じ動作となります．

なお，SH-4のコアは32ビットなので，CPUのロード/ストア命令では64ビット幅でのバスアクセスを発生させることはできませんが，内蔵されているDMAコントローラには，1ワードを64ビットに設定可能です．DMAを使って1ワードを64ビットとしたとき，バス幅32ビットに設定された空間をMPXバスモードで1ワードアクセスすると，2ワードのバースト転送が発生します．つまり，バス幅64ビットの時のバースト転送は4ワードのみで他はすべてシングル転送，32ビットの時は8ワードもしくは2ワードのバースト転送と，32/16/8ビットのシングル転送が発生することになります．

● バーストライト

図8にMPXバスモード/バーストライト時のバスの動作波形を示します．アドレス出力と1ワード目のデータ出力が連続しているので，バーストリードより1クロック短い時間でアクセスが完了します．

なお，バースト転送でもBS信号はバスアクセス開始の1クロ

ック期間のみアサートされるので，アドレスフェーズの判定はこれを使うことが可能です．

また MPX バスモードでも，同一エリアに連続してアクセスしている間は，$\overline{CS}$[*n*]はネゲートされたままとなります．

● **MPX バスモードでの注意点**

MPX バスモードはすべてのエリアで使用可能なのですが，問題はその切り替え単位です．エリア 0 は単独でほかのエリアと独立して切り替え可能ですが，それ以外のエリア 1～6 は，まとめて標準バスモードか MPX バスモードかを選択するしかありません．これは SH-4 のバスコントローラの仕様です．

ただし，CPU 内蔵の SDRAM コントローラを使い，エリア 2 と 3 に SDRAM を繋いでいる場合は，エリア 1～6 を MPX バスモードに切り替えても，エリア 2 と 3 を SDRAM モードで使用できます．

MPX バスモードでは，アドレスフェーズで確定している情報は，アドレス情報とアクセスサイズ情報，そして読み出しか書き込みかのアクセス方向のみです．

MPX バスモードではライトバイトイネーブル信号が直接出力されないので，アドレスフェーズで取得したアクセスサイズ情報とアドレス情報から，自前でライトバイトイネーブルを生成しなければならず，この信号を生成するのに最低 1 クロックが必要になると思われます．しかしよく考えてみると，実際のデータ転送のタイミングは，どんなに早くてもアドレスフェーズの次のクロックからです．よって 1 クロック以内でライトバイトイネーブルを生成することができれば，実質ノーウェイトでのアクセスも可能になります．

また，標準バスモードのエリア 1 とエリア 4 以外のエリアでは，リードアクセス時はバイトイネーブル情報が出力されませんでした．しかし，MPX バスモードではリードアクセスでもアクセス情報が出力されるので，リードバイトイネーブル情報を生成することができます．

● **バスステートコントローラ**

これらバス幅やバスの動作モードの設定，バスアイドル時やバスアクセス時のウェイト数の設定は，SH-4 のエリア 7 の内蔵レジスタ領域に配置されているバスステートコントローラを使って行います．

〔図 8〕MPX バスモード/ノーウェイト/バーストライト時のバスの動作波形

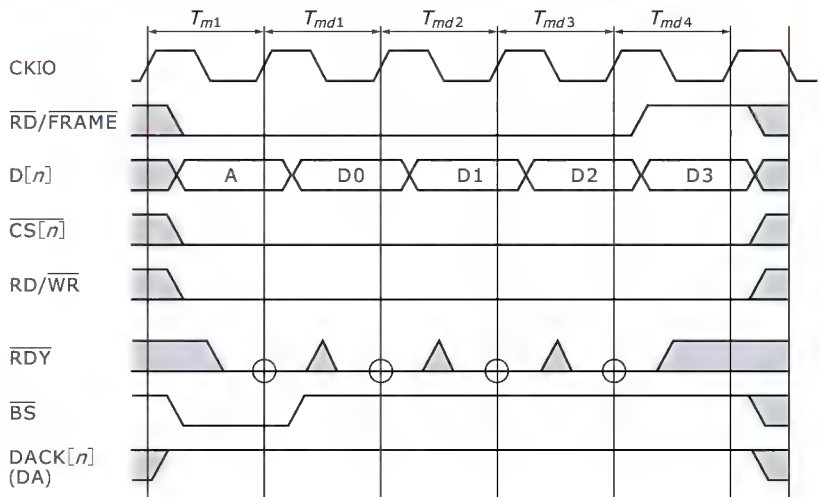



表 2 に SH-4 のバスステートコントローラのレジスタを示します．これ以外に DRAM や PCMCIA モードに関連したレジスタもありますが，本システムでは使用しません．これらのレジスタに適切な値を設定することで，外部バスの動作を設定するわけです．各レジスタの詳細は SH-4 のハードウェアマニュアルを参照してください．

2 プロセッサボードの構成

● **プロセッサボードの構成**

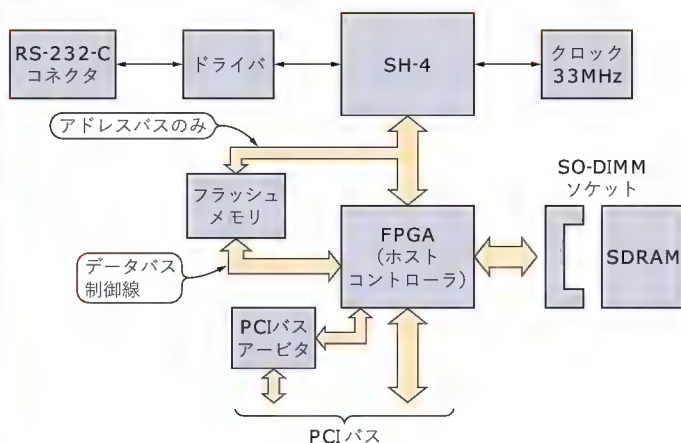
今回採用した CPU である SH7750 は PCI ホストコントローラを内蔵していないので，必然的に PCI コントローラは外付けになります．

PCI バスシステムでは，バスマスタデバイスがホスト CPU のメインメモリ領域に直接アクセスすることで，CPU 負荷を軽減したデータ転送を可能にします．正確には，バスマスタデバイスがイニシエータ，PCI ホストコントローラがターゲットとなり，PCI ホストコントローラがメインメモリにアクセスすることになります．

SH-4 にも SDRAM コントローラが内蔵されているので，それを使って SO-DIMM の SDRAM を制御することも可能です．しかし，この方法では SH-4 以外のデバイスが SDRAM にアクセス

〔表 2〕SH-4 のバスステートコントローラのレジスタ

名　称	略　称	R/W	初期値	P4 アドレス	エリア 7 アドレス	アクセスサイズ
バスコントロールレジスタ 1	BCR1	R/W	0000_0000h	FF80_0000h	1F80_0000h	32
バスコントロールレジスタ 2	BCR2	R/W	3FFCh	FF80_0004h	1F80_0004h	16
バスコントロールレジスタ 3	BCR3	R/W	0000h	FF80_0050h	1F80_0050h	16
バスコントロールレジスタ 4	BCR4	R/W	0000_0000h	FF0A_00F0h	1F0A_00F0h	32
ウェイトコントロールレジスタ 1	WCR1	R/W	7777_7777h	FF80_0008h	1F80_0008h	32
ウェイトコントロールレジスタ 2	WCR2	R/W	FFFE_EFFFh	FF80_000Ch	1F80_000Ch	32
ウェイトコントロールレジスタ 3	WCR3	R/W	0777_7777h	FF80_0010h	1F80_0010h	32

〔図9〕プロセッサボードの構成

することが難しくなります．つまりバスマスタがメインメモリに対してアクセスできなくなるのです．

そこでSDRAMコントローラも外付けで用意し，CPUからのアクセスとPCIバスマスタからのアクセスの両方に対応できるようにするわけです．

そこで，**図9**のようにFPGAを中心に置き，CPU，SDRAM，PCIをそれぞれFPGAと直結する構成にしました．また，このCPUとメモリ，PCIホストコントローラの部分までをPCIハーフサイズの1枚のボードに実装し，これをプロセッサボードと呼ぶことにします．実際のI/O機能は，第3章で説明しますが，PCIバックプレーンを使ってPCIスロットを装備し，そこにPCIボードを差し込むようにしてシステムを構成します．

写真1にプロセッサボードの外観を示します．**写真1**を見ると，コンパクトフラッシュソケットやIDEコネクタなどが見えますが，これらは別の目的で使用するためのもので，今回のシステムでは使用できません．

写真2に，プロローグに掲載したマイクロATX筐体の中を拡大した写真を示します．市販のマイクロATX筐体に，第3章で紹介するPCIバックプレーンを固定し(ネジ穴の位置が合わないので，一部筐体を加工した)，PCIスロットに各ボードを実装しています．

写真3にPCIバックプレーンにプロセッサボードや各種インターフェースPCIボードを実装したときの最小構成システムの外観を示します．3.5インチHDDの上に載っている小基板は，第5章で解説するキーボード＆マウスコントローラのためのマイコンボードです．

● フラッシュメモリの配置

SH-4のブート用であるフラッシュメモリは，FPGAのI/Oピン数の関係から，**図9**に示すようにアドレスバスをCPUから配線し，その他の制御線およびデータバスはFPGAと接続しています．よって，フラッシュメモリをアクセスするには，SH-4が標準バスモードで動作しなければなりません．MPXバスモードでの対処法は後述します．

● メモリマップ

表3にプロセッサボードのメモリマップを示します．SH-4はリセット直後，物理アドレス0000_0000hから命令を読み込み，実行をはじめます．内部的にはP2領域からアクセスするので，A000_0000hのアドレスとなります．いずれにせよ，エリア0の先頭アドレスからアクセスをはじめることになります．

SH-4のリセット解除直後の起動は，68000シリーズなどのようにリセットベクタを読み込むような動作はせず，そのアドレスから直接命令を読み込もうとします．よって，必然的にエリ

〔写真1〕プロセッサボードの外観

〔写真2〕市販マイクロATX筐体に実装したようす

〔写真3〕最小構成時のシステムの外観

ア0にはROMを配置しておくことになります.

しかし今回は,RAMを最大256Mバイト搭載することを考えています.メモリマップ的に,エリア0をROMに,エリア1～4の合計256MバイトをRAMに割り当てるという方法もありますが,それではあまりにも中途半端です.ここは2のn乗アドレスの単位からメモリをマップしたいところです.

● エリア0のバンク切り替え

そこで,ハードウェアリセット時は,エリア0にもエリア6と同じ空間をマッピングし,先頭部分にROMを配置します.リセット直後エリア0でROMを読み込み,バスの最低限の初期化を行った後,エリア6にジャンプするようにします.エリア6にジャンプしたら,エリア0のバンク切り替えレジスタを操作し,エリア0をRAMに切り替えます.

ただし,すでに説明したように,すべてのエリアを標準バスモードで使う場合は問題ありませんが,MPXの場合には,そのままでは直接フラッシュメモリをアクセスすることができません.

そこでMPXバスモードで使うときには,FPGA内部のメモリブロックを使い,そこにフラッシュメモリからRAMにデータを転送するサブプログラムを置いて実行することで対処します.

エリア0をMPXバスモードで使い,残りのエリアを標準バスモードで,またはその逆で使うというパターンはあまり想定していません(正確には全体をMPXバスモードで使う場合は,フラッシュメモリからRAMへの転送する間は前者の動作になるが).すでに説明したように,エリア0～3までを連続したRAM領域として使うことを想定しているので,エリア0とエリア1の間でバスの動作モードが異なると,標準バスモードとMPXバスモードの両方に対応したローカルバスコントローラが必要になるからです.

● SH-4の割り込みシステム

SH-4には,$\overline{\mathrm{IRL}}$[3～0]までの4本のマスク可能な通常の外部割り込み入力と,1本のマスク不可能な割り込み(NMI)入力があります.本システムではNMIは将来の拡張用に,ピンヘッダを用意して外部からの接続を可能にしておきました.

4本の割り込み入力はレベルを示し,$\overline{\mathrm{IRL}}$[3～0]がすべて"H"レベルの場合はレベル0で割り込みなしの状態と見なされ,すべて"L"レベルの場合はレベル15で最上位(NMIよりは下)レベルの割り込みとなります.

また外部に二つから四つの割り込み出力デバイスがある場合,ハードウェアを簡略化するために,4本の$\overline{\mathrm{IRL}}$にそれぞれのデバイスからの割り込み出力信号を接続する使い方も可能です.

● ホストコントローラからの割り込み出力

今回は**図9**で示したように,CPUとFPGAを直結しているので,CPUに対して直接的に割り込みを出力するのはFPGAのみとなります.そこで4本のIRLをそのままFPGAに接続し,FPGA内部にIRLレベルレジスタを用意することで,任意の割り込みレベルでCPUに対して割り込みを出力できる構成にします.

〔表3〕プロセッサボードのメモリマップ

エリア	物理アドレス	論理アドレス(P2領域)	名　称	容　量(バイト)
0	0000_0000h ～	A000_0000h ～	SDRAM/フラッシュメモリ	64M
1～3	0400_0000h ～	A400_0000h ～	SDRAM	192M
4～5	1000_0000h ～	B000_0000h ～	PCIバスメモリ空間	128M
6	1800_0000h ～	B800_0000h ～	フラッシュメモリ	16M
	1900_0000h ～	B900_0000h ～	(予約)	16M
	1A00_0000h ～	BA00_0000h ～	PCI I/O空間ウィンドウ	16M
	1B00_0000h ～ 1BFF_FFFFh	BB00_0000h ～ BBFF_FFFFh	各種制御レジスタ空間	16M

また本システムでは,メインメモリ以外のリソースをすべてPCIバス空間に接続するようにするので,割り込みの発生源になるものは,PCIバス関連しかありません.実質的にPCIバスコントローラからの割り込みがSH-4に出力されることになります.PCIバスコントローラの割り込みについては,第3章を参照してください.

3 ローカルバスコントローラの設計例

今回のシステムではCPUの信号線をすべてFPGAに直結しているので,SH-4からFPGAへのアクセス要求を制御しなければ,後述のSDRAMコントローラやPCIバスコントローラなどとも,いっさいデータの受け渡しができません.

ここでは,FPGA内部に読み書き可能なレジスタを1本用意し,このレジスタに対するSH-4からのライトサイクル時とリードサイクル時の応答を確実に行うバスコントローラの設計を行ってみましょう.

● ローカルバスコントローラ仕様

本章ではこれから続く章のはじまりとなる,最低限必要なSH-4のバスコントローラの設計を行います.まずSH-4の外部バスの動作確認のために,次のような仕様のバスコントローラを作成してみましょう.

- SH-4のバスクロックに同期した完全同期設計
- 32ビット/標準バスモードと64ビット/MPXバスモード対応
- バス幅と同一の1本のリード/ライト可能なレジスタを実装
- SH-4からのアクセスは$\overline{\mathrm{BS}}$信号によりバスサイクルの開始を検出して内部処理を行う
- レジスタはエリア0空間に実装する
- SH-4のバスクロックに同期して$\overline{\mathrm{RDY}}$ピンを駆動するハンドシェイク機能をもつ
- シングル転送のみ対応

サンプルのVHDLソースを**リスト1**(章末)に示します.

〔表4〕64ビットバス幅時のデータアライメント

動作		データバス							
アクセスサイズ	アドレス	D63～56	D55～48	D47～40	D39～32	D31～24	D23～16	D15～8	D7～0
バイト	8n	データ7～0	–	–	–	–	–	–	–
	8n+1	–	データ7～0	–	–	–	–	–	–
	8n+2	–	–	データ7～0	–	–	–	–	–
	8n+3	–	–	–	データ7～0	–	–	–	–
	8n+4	–	–	–	–	データ7～0	–	–	–
	8n+5	–	–	–	–	–	データ7～0	–	–
	8n+6	–	–	–	–	–	–	データ7～0	–
	8n+7	–	–	–	–	–	–	–	データ7～0
ワード	8n	データ15～8	データ7～0	–	–	–	–	–	–
	8n+2	–	–	データ15～8	データ7～0	–	–	–	–
	8n+4	–	–	–	–	データ15～8	データ7～0	–	–
	8n+6	–	–	–	–	–	–	データ15～8	データ7～0
ロングワード	8n	データ31～24	データ23～16	データ15～8	データ7～0	–	–	–	–
	8n+4	–	–	–	–	データ31～24	データ23～16	データ15～8	データ7～0
クワッドワード	8n	データ63～56	データ55～48	データ47～40	データ39～32	データ31～24	データ23～16	データ15～8	データ7～0

(a) ビッグエンディアン時

動作		データバス							
アクセスサイズ	アドレス	D63～56	D55～48	D47～40	D39～32	D31～24	D23～16	D15～8	D7～0
バイト	8n	–	–	–	–	–	–	–	データ7～0
	8n+1	–	–	–	–	–	–	データ7～0	–
	8n+2	–	–	–	–	–	データ7～0	–	–
	8n+3	–	–	–	–	データ7～0	–	–	–
	8n+4	–	–	–	データ7～0	–	–	–	–
	8n+5	–	–	データ7～0	–	–	–	–	–
	8n+6	–	データ7～0	–	–	–	–	–	–
	8n+7	データ7～0	–	–	–	–	–	–	–
ワード	8n	–	–	–	–	–	–	データ15～8	データ7～0
	8n+2	–	–	–	–	データ15～8	データ7～0	–	–
	8n+4	–	–	データ15～8	データ7～0	–	–	–	–
	8n+6	データ15～8	データ7～0	–	–	–	–	データ15～8	データ7～0
ロングワード	8n					データ31～24	データ23～16	データ15～8	データ7～0
	8n+4	データ31～24	データ23～16	データ15～8	データ7～0	–	–	–	–
クワッドワード	8n	データ63～56	データ55～48	データ47～40	データ39～32	データ31～24	データ23～16	データ15～8	データ7～0

(b) リトルエンディアン時

● 設計上の留意点

バスサイクル開始時にまず行わなければならないことは，$\overline{\mathrm{BS}}$がアサートされたときに，SH-4から出力されたアドレスとアクセスサイズを取り出して保持することです．

この部分はソースコードの中で，クロックの立ち上がりエッジごとに$\overline{\mathrm{BS}}$信号をサンプルして，SH-4からの各信号ラインから以下の信号を生成しています．

- アドレスバス： A[63：00]の下位26ビットを取り出してADRS[25：00]とする．さらに$\overline{\mathrm{CS}}$[6：0]よりADRS[28：26]を生成する
- アクセスサイズ： A[63：61]の3ビットを取り出してSIZE[2：0]とする

- アクセスエリア：$\overline{\mathrm{CS}}$[6：0]を保持して $\overline{\mathrm{hold_CS}}$[6：0]信号とする
- アクセス方向：RD/$\overline{\mathrm{WR}}$を保持して，hold_R_nW とする
- バイトイネーブル：転送サイズとアドレスバス A[2：0]からアクセスのあるバイト位置を特定する

ここで取り出したアドレス値やバイトイネーブルは，FPGA 内部のレジスタへのライトサイクル時に必要な信号です．とくにバイトイネーブルは，標準バスモードではバイト制御 SRAM モードを使うと SH-4 がバイト単位で出力してくれるのですが，MPX バスモードではアクセスサイズ信号として出力されるのでそのままでは使えません．MPX バスモードの場合は，バイト単位のアクセスはバスコントローラ側の責任でサポートしなければなりません．

もう一つの注意点は，エンディアンの扱いです．SH-4 はリセット時にビックエンディアンかリトルエンディアンのどちらかに設定して使用します．動作途中でエンディアンを変えることはできません．

表 4 に 64 ビットバス時のデータアライメントを示します．ビッグエンディアンのときはデータバスの上位ビット側から，リトルエンディアンのときはデータバスの下位ビット側からアドレスをカウントします．

ちなみに，SH-4 の CPU コアは 32 ビットなので，CPU のロード/ストア命令では**表 4** にあるクワッドワードアクセスは発生しません．これは内蔵の DMA コントローラで，1 ワードのサイズを 64 ビットとしたときに発生します．

● 動作説明

ここでは 64 ビット MPX バスモードでの動作について説明します．

MPX バスモードでは，まずバスサイクルの始まりを表す $\overline{\mathrm{BS}}$ のアサート（と $\overline{\mathrm{FRAME}}$ のアサート）時をアドレスフェーズ，2 クロック目以降がデータフェーズと呼びます．

まず 1 クロック目のアドレスフェーズで，$\overline{\mathrm{BS}}$ のアサートをサンプルした立ち上がりエッジで，FPGA 内部の ADRS[28：00]（アドレス保持レジスタ）やサイズ値保持レジスタ，アクセス方向レジスタなどを保持しています．

そして，2 クロック目のデータフェーズに入った段階では，まだ FPGA 内部の TEST_REG（値を保持するテストレジスタ）へのアクセスは行いません．このタイミングで，1 クロック前に保持したアドレスレジスタ（ADRS[2：0]）やアクセスサイズレジスタ（SIZE[2：0]）から，FPGA 内部の PBE[7：0]（バイトイネーブル信号）を生成しています．

3 クロック目には，FPGA 内部で「アクセスアドレス」，「バイトイネーブル」，「アクセス方向」の情報すべてがそろうので，ここでライトサイクルであれば，バス上の 64 ビットデータを FPGA 内部のテストレジスタに取り込みます．

リードサイクルであれば出力ポートにテストレジスタの値をセットし，同時にデータバスの出力イネーブル信号をアサートしてデータバスを有効にします．

$\overline{\mathrm{RDY}}$信号のアサートはこの 3 クロック目に行います．そしてアサートする期間は 1 クロック分のみです．すると SH-4 は次の 4 クロック目で $\overline{\mathrm{RDY}}$ をサンプルし，5 クロック目でバスサイクルを完了します．リードサイクルであればこの 5 クロック目のバスサイクルの完了時点で，データバス上に出力されているデータを SH-4 内部に取り込みます．FPGA 側ではこのタイミングでデータバスをドライブを開放（ハイインピーダンス処理）します．

ライトサイクルであれば，SH-4 はこのタイミングまでデータバスのデータを出力し続けます．FPGA 側はすでに書き込みデータを取り込んでいるので，何もすることはありません．

● CPU にウェイトをかけるには

この例は，FPGA 内部に用意したテストレジスタが，いつでも応答できるレジスタだったので，CPU のアクセス開始から何クロック目で何をして，次のクロックで何をする……という動作が決定できます．

しかし，これが SDRAM などの DRAM だとどうでしょうか．DRAM には内容を保持するためのリフレッシュという動作が必要です．CPU が DRAM をアクセスしようとしたときにリフレッシュ中だった場合，DRAM は CPU のアクセスにすぐには応答できません．この場合はリフレッシュが終了するまで，CPU を待たせなければなりません．CPU を待たせるには，$\overline{\mathrm{RDY}}$をアサートするタイミングを遅らせることで調整します．

● 本格的なローカルバスコントローラ

より本格的な SH-4 ローカルバスコントローラについて説明します．SH-4 ローカルバスコントローラと SDRAM コントローラ間の入出力ポートの仕様を示します．

- CLK ：動作クロック
- $\overline{\mathrm{RST}}$ ：システムリセット
- PA[28：00] ：プロセッサアドレスライン
- PDI[31：00] ：プロセッサデータ入力バッファ
- PDO[31：00] ：プロセッサデータ出力レジスタ
- PRD/$\overline{\mathrm{WR}}$ ：リード/ライト転送方向信号
- PBE[7：0] ：バイトイネーブル
- TS ：転送開始要求信号
- TA ：アクセス完了応答信号

このインターフェースを実現するためには，SH-4 によってドライブされた各制御線から，アドレスバス，ライトデータ，バイトイネーブルなどとともに，メインメモリへのアクセスの場合は SDRAM コントローラに対する転送開始要求と，SDRAM コントローラからのアクセス完了応答信号を生成する必要があります．

TS 信号（Transfer Start：転送開始要求）をアサートすることで，アクセス先の SDRAM コントローラを起動します．この信号は SH-4 の $\overline{\mathrm{CS}}$[6：0]などのうち，SDRAM をマッピングしたエリアのチップセレクト信号をそのまま使ったほうが便利でしょう．第 2 章では 256M バイトの空間を SDRAM メモリとして

使いたかったため，エリア0からエリア3までのをTS信号のアサートのために使うことにしました．

TS信号は，筆者が設計したSDRAMコントローラの仕様では1クロックの期間アサートするだけでいいようにしています(ちょうどSH-4の$\overline{\mathrm{BS}}$信号と同じ仕様)が，世の中に出回っている多数のIPコアでは，チップセレクトは転送完了になるまで出し続けるように要求するものが多いようです．

汎用的に使えるものを考えるため，ここではとくに筆者が設計したSDRAMコントローラだけにこだわらず，TSはTAがアサートされるまで出し続けるような仕様にしました．

また，もう一つのTA信号(Transfer Acknowledge：アクセス完了応答信号)については，この信号がアサートされたことで，SH-4側へ$\overline{\mathrm{RDY}}$信号を出すことになります．本誌で想定したSDRAMコントローラの仕様では，ライトサイクル時にTAがアサートされたらSDRAMにライトサイクルが無事に発行されたことを示し，リードサイクル時にTAがアサートされたらその時点でPDO[31：00]バスには読み出したデータがセットされていることを表します．

ただし，SH-4の$\overline{\mathrm{RDY}}$信号の認識とバスアクセス完了のタイミングに注意してください．SH-4は$\overline{\mathrm{RDY}}$信号のアサートをサンプルした次のクロックでバスサイクルを終了し，リードサイクルであればデータバス上のデータを取り込みます．よってSDRAMコントローラからは，データ出力と同時にTAがアサートされて転送完了を表していますが，SH-4へのPDO[31：00]の出力は，$\overline{\mathrm{RDY}}$をアサートしても，さらに1クロック期間だけはデータバスをドライブしなければなりません．

まとめ

バスアクセスの動作をステートマシンで制御するは常道です．しかし多少複雑な動作をするバスになると，ステートマシンが複雑になってくるのは避けられません．また64ビット幅のデータをレジスタにロードするには，VHDLでは単純に，

```
TEST_REG(63 downto 0) <= SH_D(63 downto 0);
```

と記述しますが，これを，バス制御の中のどのステートの段階でロードするのかを理解しやすいように，ステートマシンの中に記述してしまうと，長大なレジスタをステートマシン内で制御するため，ファンアウト(信号線の接続先)が多いために，配線による遅延時間が膨大になり，なかなか高速で動作してくれないという問題に突き当たります．

そこで筆者のところでは，ステートマシンをできるだけ小さく分割したり，サイズの大きなレジスタはステートマシンの外に出し，さらにいくつかに分割しています．そしてステートマシン中から制御する数ビットの内部信号線にしたがってリード/ライト制御させるなどの方法を使っています．

このような最適化をすることで，今回のこの同じ基板で同じグレードのFPGAで，SH-4のローカルバスを100MHzで動作させています．

最近の論理合成エンジンはかなり賢くなってきているとはいえ，そこまでの最適化はなかなかしてくれません．採用するデバイスのマクロセルの構造を考えて，その構造で最適な入力数になるようにHDLソースを記述する段階から分割して記述するなど，人間による最適化の手間をかけないと，100MHzや133MHzといったクロックで動作する回路は実現できません．また，そのようなソースは，ステート動作の全体が追いにくい，読みにくいソースになってしまうこともあります．

もちろん，コストがかかることを承知で，もっと高速に動作するグレードの高いデバイスを使えば，何も考えずに記述しても100MHzに達します．しかし，安いデバイスを使って少しでも速い回路を実現するには，まだまだそれなりの苦労とノウハウが必要になります．

参考文献

1) TECH I Vol.1『SuperHプロセッサ』，CQ出版(株)
2)『SH7750シリーズ ハードウェアマニュアル』第6版，(株)日立製作所

いくら・まさみ　来栖川電工有限会社

〔リスト1〕SH-4用ローカルバスコントローラのVHDLソース(一部)

```
library IEEE;
use IEEE.STD LOGIC 1164.ALL;
use IEEE.STD LOGIC ARITH.ALL;
use IEEE.STD LOGIC UNSIGNED.ALL;

entity HORNET64 is
    port (
                                ～中略～
    );

end HORNET64 ;

architecture Behavioral of HORNET64 is

-- ************************************************************ --
    signal    nRST       : std logic ;
    signal    pRST       : std logic ;
    signal    OpCLK           : std logic ;
    signal    sync nPBS       : std logic ;
    signal    sync nPCS       : std logic vector(6 downto 0) ;
    signal    sync nPFRAME    : std logic ;
    signal    sync PR nW           : std logic ;
    signal    sync nPWE       : std logic vector(7 downto 0) ;
    signal    sync PAD        : std_logic_vector(63 downto 0) ;

-- ************************************************************ --
                                ～中略～
-- ************************************************************ --
    signal    PAD_O_Port      : std_logic_vector(63 downto 0) ;
    signal    PAD_O_Reg       : std_logic_vector(63 downto 0) ;
    signal    nPREADY_O_Port : std_logic ;

    signal    PAD O OEN node : std logic ;
    signal    PAD O OEN Port : std logic vector(63 downto 0) ;

-- ************************************************************ --
component MPXSDRAMC is
```

〔リスト 1〕SH-4 用ローカルバスコントローラの VHDL ソース(一部)(つづき)

```
    port (
    -- ******** SH4 bus interface ******** --
          BUSCLK         : in std logic ;
          nRESET         : in std_logic ;

          PAD I          : in std logic vector(63 downto 0) ;
          PAD O          : out     std logic vector(63 downto 0) ;
          PAD O OEN      : out     std logic ;

          nPBS      : in std logic ;
          nPCS      : in std_logic_vector(3 downto 0) ;
                                                -- area chipselect.
          nPFRAME        : in std logic ;
-- Access start
          PR nW          : in std logic ;
          nPWE      : in std_logic_vector(7 downto 0) ;
                                             -- write byte enable.

          SDRAMC CS : in std logic ;
          nPREADY        : out     std logic ;
                                          -- READY* output.
                                          -- 0 : READY / 1 : WAIT

    -- **** SDRAM signals **** --
          PAD_MA         : out     std logic vector(13 downto 0) ;
                                          -- SDRAM address.
                                          -- RAS-Adrs : MA[10:00]
                                          -- CAS-Adrs : MA[07:00]
          PAD MBA        : out     std logic vector( 1 downto 0) ;
                                          -- BANK address.
          PAD MD         : inout   std logic vector(63 downto 0) ;

          PAD nMCS  : out     std logic ;          -- CS*
          PAD nMRAS : out     std logic ;          -- RAS*
          PAD_nMCAS : out     std logic ;          -- CAS*
          PAD nMWE  : out     std logic ;          -- WE*
          PAD_DQM        : out     std logic vector( 7 downto 0) ;
                                                      -- DQM[7:0]

    -- **** For DEBUG function **** --
          SDRAMC Status  : out     std logic vector(15 downto 0)

    ) ;
end component;

    signal    SDRAMC CS      : std logic ;
    signal    SDRAMC Status  : std logic vector(15 downto 0) ;

begin

                                   ~中略~

-- ************************************************************ --
SH4IF Sync : process ( OpCLK )
begin
    if ( OpCLK'event and OpCLK = '1' ) then
          sync nPBS      <= nPBS ;
          sync nPCS      <= nPCS    ;
          sync nPFRAME   <= nPFRAME ;
          sync nPWE      <= nPWE ;
          sync PR nW          <= PR nW ;
          sync PAD       <= PAD ;
    end if ;
end process ;
-- ************************************************************ --

PAD Output equ : process ( OpCLK )
begin
    if ( OpCLK'event and OpCLK = '1' ) then
          PAD O reg <= PAD O Port ;
    end if ;
end process ;

PAD OEN Equ1 : process ( OpCLK, nRST )
begin
    if ( nRST = '0' ) then
          PAD O OEN node2     <= ( others => '0' ) ;
    elsif ( OpCLK'event and OpCLK = '1' ) then
          u4 : for i in 0 to 7 loop
               PAD O OEN node2(i)  <= PAD_O_OEN_node ;
                                          -- 1=>8 bus width translate
          end loop ;
    end if ;
end process ;

u6 : for i in 0 to 63 generate
    PAD(i)    <= PAD O reg(i)     when PAD O OEN Port(i) = '1'
                                                      else 'Z' ;
end generate ;

-- ************************************************************ --
SDRAMC CS equ : process ( OpCLK, nRST )
begin
    if ( nRST = '0' ) then
          SDRAMC CS <= '0' ;
    elsif ( OpCLK'event and OpCLK = '1' ) then
          if ( nPBS = '0' and nPCS(3 downto 0) /= "1111" ) then
               SDRAMC CS <= '1' ;
          else
               SDRAMC CS <= '0' ;
          end if ;
    end if ;
end process ;
-- ************************************************************ --

nREADY Equ : process ( OpCLK, nPBS, nRST )
begin
    if ( nRST = '0' ) then
          nPREADY   <= '0' ;
    else
          if ( nPBS = '0' ) then
               nPREADY   <= '1' ;
          elsif ( OpCLK'event and OpCLK = '1' ) then
               if ( nPREADY O Port = '0' ) then
                    nPREADY        <= '0' ;
               end if ;
          end if ;
    end if ;
end process ;

-- ************************************************************ --
uSDRAMC : MPXSDRAMC
    port map (
    -- **** BUS I/F signals **** --
          BUSCLK    => OpCLK ,

          nRESET    => nRST ,

          PAD I     => sync PAD ,
          PAD_O     => PAD_O Port ,
          PAD O OEN => PAD O OEN node ,

          nPBS      => sync nPBS ,
          nPCS      => sync nPCS(3 downto 0) ,
          nPFRAME   => sync_nPFRAME ,
          PR nW     => sync PR nW ,
          nPWE      => sync nPWE ,

          SDRAMC CS => SDRAMC CS ,
          nPREADY   => nPREADY_O Port ,

    -- **** SDRAM I/F **** --
          PAD MA    => MA ,
          PAD MBA   => MBA ,
          PAD MD    => MD ,

          PAD nMCS  => nMCS ,                     -- CS*
          PAD_nMRAS => nMRAS ,                    -- RAS*
          PAD_nMCAS => nMCAS ,                    -- CAS*
          PAD nMWE  => nMWE ,                     -- WE*
          PAD DQM   => DQM ,                      -- DQM[7:0]

    -- **** SDRAM I/F **** --
          SDRAMC_Status  => SDRAMC_Status

    ) ;

-- ************************************************************ --
--  For debug mode
--
    CLKOUT         <= OpCLK ;
    IOD(0)         <= SDRAMC Status(0) ;
    IOD(1)         <= SDRAMC Status(1) ;
    IOD(2)         <= SDRAMC Status(2) ;
    IOD(3)         <= SDRAMC Status(3) ;
    IOD(4)         <= nPREADY2 ; -- SDRAMC_Status(4) ;
    IOD(5)         <= SDRAMC Status(5) ;
    IOD(6)         <= SDRAMC_Status(6) ;
    IOD(7)         <= SDRAMC Status(7) ;
    IOD(15 downto 8)    <= nPWE(7 downto 0) ;

end Behavioral;
```

第2章

メインメモリとしてSDRAM SO-DIMMを実装する

SDRAMコントローラの設計/製作

井倉将実

ホストCPUであるSH-4にはSDRAMコントローラも内蔵されている．しかし，今回はSDRAM制御方法を学習するという点と，PCIバスマスタ対応システムを実現するという2点から，SDRAMコントローラを外付けで実現する．SDRAMデバイスおよびSO-DIMMの構成や動作を解説したあと，FPGAでSDRAMコントローラを実現する．

（編集部）

はじめに

第1章で述べたように，SH-4はチップセレクト信号で選択される空間が七つあり，各エリアは最大64Mバイトの空間があります．つまりSH-4は，外部に最大448Mバイトの物理アドレス空間をもつことになります．

今回の特集のために設計したシステムでは，この空間のうち前半の256Mバイト，エリア0からエリア3までの4エリアをメインメモリとして割り当て，SDRAMを配置するRAM領域として確保します．

このSDRAMコントローラは，前章で解説したSH-4ローカルバスコントローラや次章で解説するPCIホストコントローラとともに，ホストコントローラであるFPGAに実装します．

この章ではSDRAMメモリデバイスそのもの，およびそれをモジュール化したSO-DIMMの構造や使い方，そしてそれを応用したSDRAMコントローラの設計方法を解説します．

1 SDRAMメモリとSO-DIMMモジュール

● SDRAMの内部ブロック

図1に，一般的なPC100対応，つまりクロック周波数100MHz対応の，2Mビット×4バンク×16ビット構成の128Mビット SDRAM（HY57V1291620TC-8：旧HYUNDAI，現hynix）の内部構造を示します．

SDRAM以前に使われていた通常のDRAM（以降，非同期DRAM）では，一つのメモリブロックが一つのアドレスバスやセンスアンプに接続されていましたが，SDRAMの場合は128Mビットのメモリであるにもかかわらず，内部的には32Mビットのメモリが四つ入ったイメージになります．

また図中の左側にある“ステートマシン”とは，外部からのクロックに同期して複数の制御線を解釈し，動作を決定するコントローラです．つまり内部に高性能なコントローラが内蔵された複数バンクのメモリセルをもつDRAMモジュールというのがSDRAMのイメージです．

複数バンクに分割されているので，コマンドによるアクセス方式により，バンクごとに独立したコントロールが可能です．よって非同期DRAMと比べてバンクごとにインターリーブ動作が可能になり，見かけ上のデータ転送能力を向上させることができます．

● 各信号線の意味

SDRAMには非同期DRAMと似た名称の信号線や，新たに追加された信号線が存在します．SDRAMの各信号線について説明します．

▶ CLK（クロック）/CKE（クロックイネーブル）

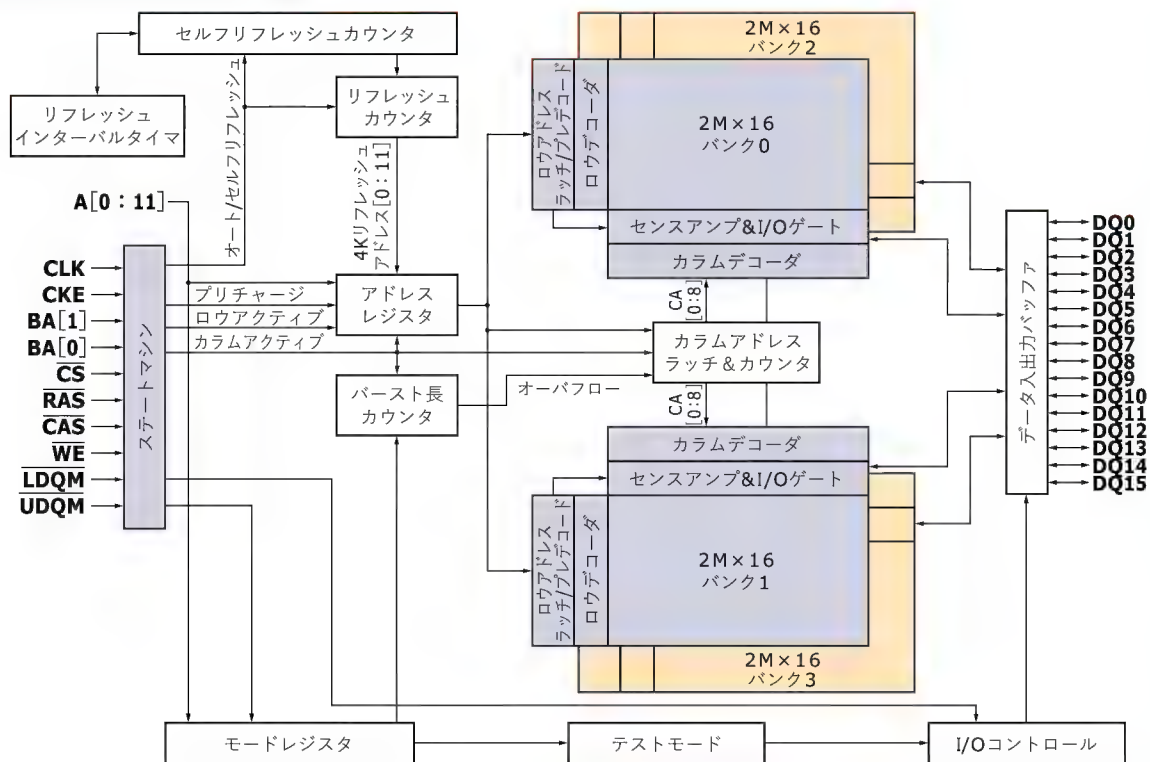
CLK信号はSDRAMを動作させるための基準クロックで，すべての制御信号線はこのクロックの立ち上がりエッジで内部に取り込まれ，デコードされて動作が決定します．

CKE信号はクロックイネーブル信号で，SDRAM内部のロジックに対してクロックが有効であることを指示する信号線です．SDRAMは低消費電力動作をサポートしていて，CKE信号を“H”にアサートしている間はCLK信号に有効なクロックを供給する必要があります．CKE信号をCKE信号が“L”にネゲートすると，SDRAMはスリープ（待機モード）に入り，低消費電力モードに移行するので，クロック供給を停止することができます．スリープモードから復帰するには，CKEに“H”をアサートするより少なくとも3クロック前からCLK信号を印可する必要があります．

▶ ADRS[n]（アドレスバス）/DQ[n]：データバス

ADRS[n]はアドレスバスで，メモリの読み書き時には行（ロウアドレス）と列（カラムアドレス）を与えるアドレスバスです．非同期DRAMと同様に，二つのアドレスは時分割（マルチプレクス）されて，CLK信号の立ち上がりエッジに同期してSDRAMに入力します．

また，SDRAMの動作の部分で説明する“モード設定”という

〔図1〕代表的なSDRAMの内部ブロック図

SDRAM固有の動作では，SDRAMの動作方法を指定する信号線として利用されます．

DQ[*n*]はデータバスで，ワード幅の分だけビット数があります．

▶ BA[*n*](バンクアドレス)

ブロック図からもわかるとおり，SDRAMはいくつかのメモリブロックが集まって一つのメモリチップとして機能します．バンクアドレス信号は，要求されたオペレーションがどのメモリブロックに対して発行するかを指定するときに利用されます．たとえば，**図1**にある4バンクのうちバンク1に対しての読み出しを指定する場合には，BA[1:0]には"01"を指定します．

▶ $\overline{\mathrm{CS}}$(チップセレクト)

デバイスのセレクト信号です．$\overline{\mathrm{RAS}}$/$\overline{\mathrm{CAS}}$/$\overline{\mathrm{WE}}$信号は$\overline{\mathrm{CS}}$がアサートされているときに有効で，クロックの立ち上がりエッジで内部コントローラに取り込まれます．通常のDRAMと同じように，チップセレクト信号がアサートされていない場合においては，CLK/CKE信号以外の入力は無視されます．

▶ $\overline{\mathrm{RAS}}$(バンクセレクト/ロウアドレスストローブ)

非同期DRAMの$\overline{\mathrm{RAS}}$(ロウアドレスストローブ)としての使い方としては，BA[*n*]によってバンクアドレスで指定されたメモリバンクのうち，どのロウアドレスを有効にするかを指定する際に利用されます．

また，$\overline{\mathrm{RAS}}$/$\overline{\mathrm{CAS}}$/$\overline{\mathrm{WE}}$の組み合わせによって，リフレッシュ要求や内部モードレジスタへのアクセス要求，バースト転送中のバーストアボート要求，プリチャージなどのさまざまな動作を決定します(さまざまな機能の詳細については後述)．

▶ $\overline{\mathrm{CAS}}$(カラムアドレスストローブ)

非同期DRAMの$\overline{\mathrm{CAS}}$(カラムアドレスストローブ)としての使い方としては，まずはじめにBA[*n*]と$\overline{\mathrm{RAS}}$によってメモリバンク/ロウアドレスが指定され，その後にカラムアドレスを指定する際に利用されます．

SDRAMの場合，連続したデータ転送(バースト転送)を行う場合には，はじめの1ワード目のロウ/カラムアドレスを指定するだけで，あとはCLKが入るたびにSDRAM動作モードレジスタに指定されたデータ転送長分だけデータの読み書きが行われます．非同期DRAMのスタティックカラムモードの変則的なものと考えることもできます．

また$\overline{\mathrm{RAS}}$と同様に，$\overline{\mathrm{RAS}}$/$\overline{\mathrm{CAS}}$/$\overline{\mathrm{WE}}$の組み合わせによってカラムアドレスストローブ以外にも使われます．

▶ $\overline{\mathrm{WE}}$(ライトイネーブル)

メモリアクセスの際，$\overline{\mathrm{CAS}}$のアサートと同時に$\overline{\mathrm{WE}}$もアサートされていると，現在のメモリアクセスは書き込みサイクルであると認識され，指定されたメモリバンク/ロウ/カラムアドレスが示すメモリセルにデータが書き込まれます．当然のことながら，$\overline{\mathrm{WE}}$がネゲートされているのであればメモリアクセスは読み出しサイクルとして扱われ，DQ[*n*](データバス)からデータが出力されます．

$\overline{\mathrm{CAS}}$と同じく1ワード目のデータ転送時だけに$\overline{\mathrm{WE}}$がアサートされていれば，以降はモードレジスタで指定された転送デー

タ長分だけ内部で書き込みサイクルが発生します．

▶ $\overline{DQMB}$（データマスク制御）

一般的なSDRAMのデータ幅構成は8/16/32ビット幅です．メモリアクセスはバイト単位でも行うので，必要なバイト位置を指示するための信号としてDQMBが使われます．DQMB信号は8ビット単位で用意されています．つまり8ビット幅SDRAMデバイスの場合は1本，16ビット幅SDRAMの場合は2本，32ビット幅SDRAMの場合は4本のDQMB制御線が存在します．

● **SDRAMの基本的な動作**

SDRAMの基本的な動作タイミングを**図2**に，SDRAMへのコマンドを**表1**に示します．SDRAMはクロック（CLK）に同期してコマンドを与える形式のメモリであり，制御信号線やアドレス信号線はすべてCLK信号に同期しています．

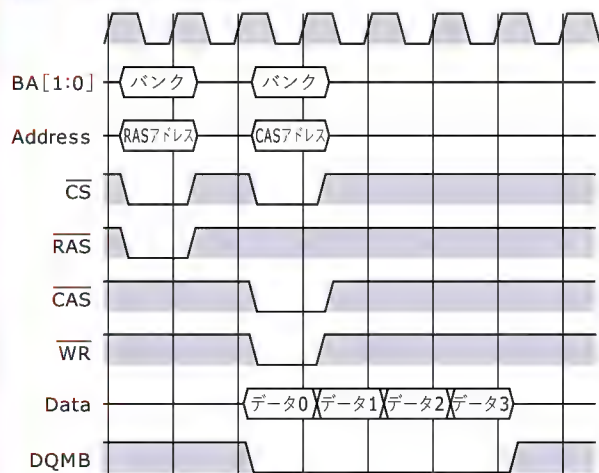
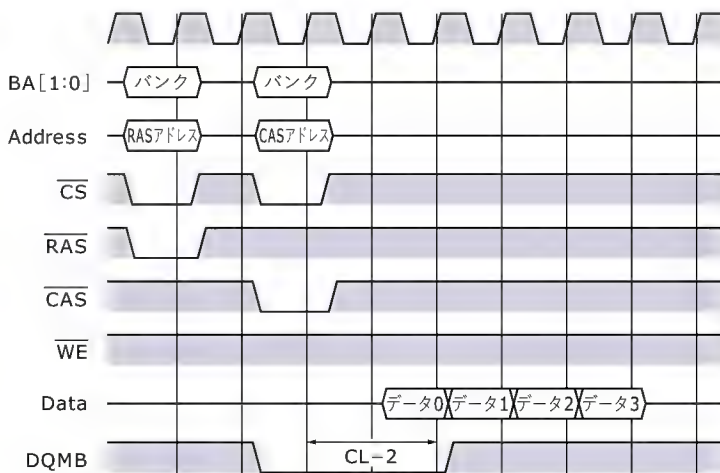
SDRAMのアクセス手順は，まずアクセスするメモリバンクとロウアドレスを有効（アクティブバンクコマンド）にして，次にアクセス先カラムアドレスとデータの転送方向を与える方式（アクセスコマンド）が基本になります．同一メモリバンク内で他のロウアドレスをアクセスするには，いったんバンクをクローズして（プリチャージコマンド）再度アクティブバンクを発行します．

プリチャージコマンドを別途与えるのが面倒な場合には，プリチャージ付きリード/ライトコマンドを発行すれば，データ転送終了時に内部で自動的にプリチャージが行われます．

図2を見ればわかるように，いったんアクティブにしたバンク/ロウアドレスに対して，連続したデータ転送時には再度アクティブを行う必要はありません．そしてSDRAMの場合は4ワード，8ワード，そしてカラムアドレス幅分のデータ転送長であるフルページバースト転送によるデータ転送を行うことが可能です．

SDRAMのスペックでCL値というものがあります．CL値とは，CAS-Latencyの略で，SDRAM内のシーケンサがアクセスコマンドを認識してから何クロック目で読み出しデータが確定するかを示すものです．データの書き込み時には影響ありません．

図2（**b**）の例でもわかるとおり，CL値が2のSDRAMの場合，$\overline{CAS}$がアサートされたアクセスコマンドをCLKの立ち上がりエッジで取り込んでから読み出しデータが出力されるには2クロックかかっています．そして3クロック目には2ワード目の値が，4クロック目には3ワード目の値，最後に5クロック目には4ワード目の値が出力され，次のクロックでハイインピーダンス

〔図2〕SDRAMの動作波形

（**a**）メモリライト

（**b**）メモリリード

〔表1〕**SDRAMのコマンド**

コマンド	シンボル	CKE	$\overline{CS}$	$\overline{RAS}$	$\overline{CAS}$	$\overline{WE}$	BA	AP	ADRS
コマンド無視	DESL	"H"	"H"	x	x	x	x	x	x
ノーオペレーション	NOP	"H"	"L"	"H"	"H"	"H"	x	x	x
バーストストップ	BST	"H"	"L"	"H"	"H"	"L"	x	x	x
カラムアドレスセット/リードコマンド	READ	"H"	"L"	"H"	"L"	"H"	V	"L"	CAS
オートプリチャージ付きリード	READA	"H"	"L"	"H"	"L"	"H"	V	"H"	CAS
カラムアドレスセット/ライトコマンド	WRIT	"H"	"L"	"H"	"L"	"L"	V	"L"	CAS
オートプリチャージ付きライト	WRITA	"H"	"L"	"H"	"L"	"L"	V	"H"	CAS
アクティブバンク/ロウアドレスセット	ACTV	"H"	"L"	"L"	"H"	"H"	V	V	RAS
対象バンクのプリチャージ	PRE	"H"	"L"	"L"	"H"	"L"	V	"L"	x
全バンクプリチャージ	PALL	"H"	"L"	"L"	"H"	"L"	x	"H"	x
リフレッシュ	REF/SELF	"H"	"L"	"L"	"L"	"H"	x	x	x
モードレジスタセット	MRS	"H"	"L"	"L"	"L"	"L"	V	V	V

$\overline{CAS}$：カラムアドレス[8：0]，$\overline{RAS}$：ロウアドレス[11：0]，V：有効データ，x：Don't care

状態に移行します．

すなわち，CL=2のSDRAMデバイスをコントロールする場合は，$\overline{CAS}$アサートから2クロック後の立ち上がりエッジで取り込んだデータが，確定した1ワード目のデータということになります．同時にSDRAMはこのクロックエッジで2ワード目データに出力ピンのドライブを切り替えるというわけです．

● SO-DIMMとは

DRAMを複数実装し，パソコンなどにおけるメモリの交換/増設を容易にするものとして，SIMMやDIMMといったメモリモジュールがあります．今回は実装スペースなどの関係から，サイズの小さなメモリモジュールが必要です．そこで，ノートパソコンなどで使われている144ピンSO-DIMMを採用しました．

写真1にSO-DIMMの外観を，**図3**に寸法図を，**表2**にピン配置を示します．SO-DIMMはさきほど説明したSDRAMメモリを複数個基板上に実装したものです．クロックラインや$\overline{RAS}$/$\overline{CAS}$/$\overline{WE}$などの信号線を共通化し，データバス幅を64ビット(ECC付きの場合は72ビット)としたメモリモジュールです．もちろんバイト単位でデータの読み書きが行えるように，DQMB[7:0]信号が用意されています．

またCB[*n*]はECC用の情報を保持するビットで，ECC対応のモジュールにだけ実装されています．$\overline{S}$[*n*]はSDRAMの$\overline{CS}$に接続されています．

● SO-DIMMの構造

SO-DIMMメモリはノートパソコン用のメモリとして広く使われており，さまざまなメーカー製のものが秋葉原などでも容易に入手できます．また64M/128M/256Mバイト，最近では1枚で512Mバイトや1GバイトのSO-DIMMもあるようです．同じ容量のモジュールなのに，実装されているデバイスの数が違うものも見うけられます．

SDRAMにはデータバス幅が8/16/32ビットの各種があるので，64ビット幅にするにはその分だけ並列に並べなければなりません．さらにモジュールは表面と裏面が使えるので，たとえば16ビット幅のSDRAMを表面だけに四つ並べて64Mバイト〔**図4(a)**〕，裏面に同様に並べて128Mバイト〔**図4(a)**〕という構成も可能です．この**図4**の例では，表面と裏面を選択する信号が$\overline{S}$[*n*]で，これをバンクセレクト信号と呼びます．これは，使う制御線の構成や本数が変わってくることを意味します．容量の違いは単純にアドレス線の上位が増えているだけではありません．

よって汎用メモリコントローラは，SO-DIMMの仕様にあわせて制御方法を動的に変更する必要があります．システムは人間の

〔写真1〕SO-DIMM外観

〔図3〕SO-DIMMの外形寸法例

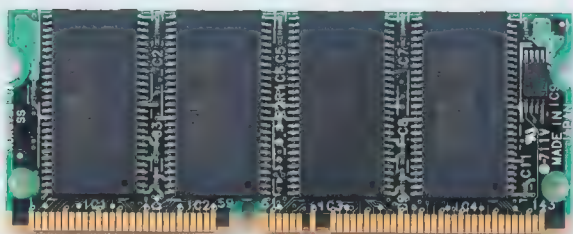
(a) 前面　(b) 横面

〔表2〕SO-DIMMのピン配置

前面ピン	信号名	背面ピン	信号名	前面ピン	信号名	背面ピン	信号名
1	V_{SS}	2	V_{SS}	73	NU	74	CK1
3	DQ0	4	DQ32	75	V_{SS}	76	V_{SS}
5	DQ1	6	DQ33	77	CB2	78	CB6
7	DQ2	8	DQ34	79	CB3	80	CB7
9	DQ3	10	DQ35	81	V_{DD}	82	V_{DD}
11	V_{DD}	12	V_{DD}	83	DQ16	84	DQ48
13	DQ4	14	DQ36	85	DQ17	86	DQ49
15	DQ5	16	DQ37	87	DQ18	88	DQ50
17	DQ6	18	DQ38	89	DQ19	90	DQ51
19	DQ7	20	DQ39	91	V_{SS}	92	V_{SS}
21	V_{SS}	22	V_{SS}	93	DQ20	94	DQ52
23	DQMB0	24	DQMB4	95	DQ21	96	DQ53
25	DQMB1	26	DQMB5	97	DQ22	98	DQ54
27	V_{DD}	28	V_{DD}	99	DQ23	100	DQ55
29	A0	30	A3	101	V_{DD}	102	V_{DD}
31	A1	32	A4	103	A6	104	A7
33	A2	34	A5	105	A8	106	BA0
35	V_{SS}	36	V_{SS}	107	V_{SS}	108	V_{SS}
37	DQ8	38	DQ40	109	A9	110	BA1
39	DQ9	40	DQ41	111	A10/AP	112	A11
41	DQ10	42	DQ42	113	V_{DD}	114	V_{DD}
43	DQ11	44	DQ43	115	DQMB2	116	DQMB6
45	V_{DD}	46	V_{DD}	117	DQMB3	118	DQMB7
47	DQ12	48	DQ44	119	V_{SS}	120	V_{SS}
49	DQ13	50	DQ45	121	DQ24	122	DQ56
51	DQ14	52	DQ46	123	DQ25	124	DQ57
53	DQ15	54	DQ47	125	DQ26	126	DQ58
55	V_{SS}	56	V_{SS}	127	DQ27	128	DQ59
57	CB0	58	CB4	129	V_{DD}	130	V_{DD}
59	CB1	60	CB5	131	DQ28	132	DQ60
61	CK0	62	CKE0	133	DQ29	134	DQ61
63	V_{DD}	64	V_{DD}	135	DQ30	136	DQ62
65	$\overline{RAS}$	66	$\overline{CAS}$	137	DQ31	138	DQ63
67	$\overline{WE}$	68	CKE1	139	V_{SS}	140	V_{SS}
69	$\overline{S0}$	70	A12	141	SDA	142	SCL
71	$\overline{S1}$	72	A13	143	V_{DD}	144	V_{DD}

〔図4〕SO-DIMMのデバイス実装例

(a) 1バンク構成64Mバイト

(b) 2バンク構成128Mバイト

ようにパッケージ上の型番などを見ることはできないので，そのままではモジュールの容量や構成を識別することはできません．

● メモリの素性を格納するメモリ～SPD～

そこでSO-DIMMでは，自身の仕様情報を保持したROMを必ず1個実装しています．これをSPD(Serial Presence Detect) ROMと呼びます．SPDはフィリプスが権利を保有するI^2Cと呼ばれる2線式シリアル通信方式を採用した8ピンのシリアルROMであり，この中にメモリモジュールに関するさまざまなデータが書き込まれております．

これらのデータは大別すると，SO-DIMMの仕様，実装されているSDRAMの仕様，各社独自のデータに分類されます．

ここで，エルピーダの256Mバイト/2バンク構成のSO-DIMM EBS26UC6APSを例にとって解説をしましょう．これは32Mワード×64ビット×2バンクという構成で，モジュール上にはEDS2516APTA(256MビットSDRAM)が8個実装されています．このモジュールのSPDに書かれている情報から，そのメモリモジュールのメモリ容量を割り出す場合，次の情報を取得します．まず，SO-DIMMの仕様から，

+03h：RASアドレス幅＝13ビット

+04h：CASアドレス幅＝9ビット

+05h：モジュールバンク数＝2バンク

+06h：メモリデータバス幅＝64ビット

次にSDRAMの仕様から，

+17h：SDRAMバンク数＝4バンク

これにより，このメモリモジュールの総メモリ容量は，

2^{RAS}幅×2^{CAS}幅×バンク数×SDRAMバンク数×8(64ビット)

＝268435456バイト＝256Mバイト

(ただし1Kバイトは1024バイト，1Mバイトは1024バイトと換算)

で計算できます．また，+31hの各バンクのメモリ容量数と+05hのモジュールバンク数を掛けても総容量は出てきます．

さらにSDRAMの仕様として，CL(クロックレイテンシ)値やt_{RCD}(RAS to CASディレイ)などの固有パラメータもわかります．

+18h：CL＝2もしくは3

+29h：t_{RCD}＝15ns，または16ns

+30h：t_{RP}＝45ns，もしくは48ns

+32h/+34h：入力信号のセットアップ時間

+33h/+35h：入力信号のホールド時間

これらのデータから，そのメモリモジュールの素性が判定でき，SDRAMメモリコントローラの仕様を決めることができます．

なお，SPDのフォーマットは168ピンのDIMMでも同様です．

● 汎用的なメモリコントローラ

PC/AT互換機などに搭載されているチップセットのSDRAMコントローラはかなり汎用的に作られており，BIOSの初期化中にSPDを読み出してRAS/CAS幅，CL値やt_{RP}などの情報を取得し，チップセットのSDRAMコントローラをセットアップしていきます．こうすることで，64Mバイトでも1Gバイトでも，また同一容量でバンク数が異なる場合であっても対応することが可能になります．

より具体的な例として，同じ256Mバイトの容量で1バンク構成と2バンク構成という二つのSO-DIMMを例にして，アドレスセットアップ部分についての回路を構成した場合を説明します．まず1バンク構成のSO-DIMM(EBS25EC8APSA：エルピーダ)の場合は，

- モジュールバンク数＝1
- RASアドレス幅＝13ビット
- CASアドレス幅＝10ビット
- メモリ自身のバンク数＝4バンク(これはBA[1:0]にあたる)

2バンク構成のSO-DIMM(EBS26UC6APS：同)の場合は，

- モジュールバンク数＝2
- RASアドレス幅＝13ビット
- CASアドレス幅＝9ビット

- メモリ自身のバンク数＝4バンク(これはBA[1：0]にあたる)

となります．

1バンク構成のメモリの場合は，$\overline{S}$[0]信号のみがメモリに接続されており，2バンク構成のメモリの場合にはバンク1に$\overline{S}$[0]が，バンク2に$\overline{S}$[1]が接続されていて，それぞれ選択されるというわけです．

次は，RAS/CASアドレスとメモリ自身のバンクアドレスの割り当てです．

- RASアドレスには同一本数の13本
- CASアドレスは1バンク構成品，2バンク構成品ともに9本までは共通
- バンク数によりCASのアドレス1本がかわる

このようなルールに当てはめ，筆者流にアドレスを割り当てた例は次のようになります．

- 64ビット幅なのでA[02：00]は未使用
- A[11：03]をCAS[08：00]に割り当てる
- A[24：12]をRAS[12：00]に割り当てる
- A[26：25]をBA[1：0]に割り当てる

以上はどちらも共通で，

- 1バンク構成のときはA[27]をCAS[09]にして，$\overline{S}$[0]は常時アサート(メモリアクセスのある時のみアサート)
- 2バンク構成のときはA[27]を$\overline{S}$[0]に，A[27]の反転を$\overline{S}$[1]に割り当てる

(LSB側がA[00]，MSBアドレス側がA[31]とする)

としてみました．アドレス範囲もA[27：00]の28ビット長ですから，正しく256Mバイトの空間が使われていることになります．

PC/AT互換機などに搭載されている汎用的なSDRAMメモリコントローラは，ここで説明したことを，下は数十Mバイトから上はGバイトオーダまでの間のさまざまなメモリで対応しているのです．

2 SDRAMコントローラの設計

今回設計したSDRAMコントローラは，読者のみなさんに理解しやすいよう，多少冗長なコードになるのを承知で設計しているため，メモリアクセス性能は正直いってかなり劣るコントローラです．しかし，まずはSDRAMの基本的な制御方法をマスタするのが第一歩です．

実装するFPGAは，Spartan-II/200-5FG456C(ザイリンクス)で，このSDRAMCコントローラはSH-4の外部バスクロックと同一速度で動作させます．今回のシステムではSH-4の外部バスクロックを66MHzとしているので，このSDRAMコントローラも66MHzで駆動します．

● 汎用SDRAMコントローラの難しさ

今回設計するSDRAMコントローラは，特定の容量のSO-DIMMに限定したものとします．その理由は大きく分けて二つあります．

まず一つは，SPDに使われているシリアル通信方式が，フィリップスの特許を採用したものであり，このアクセスを行う回路をFPGAなどにハードウェアとして実装する場合，そのつどフィリップスに確認を取らなければならないとされています．残念ながら自由に使ってよいという仕様ではないのです．

次にコントローラをプログラマブルに設計するということは，回路規模が大きくなることを意味します．さらにFPGAでは，出力段が多段のセレクタ/マルチプレクサ構成になってしまい，信号の遅延時間が大きくなってきます．これはそのまま，高速動作させることが難しくなることを意味します(**図5**)．

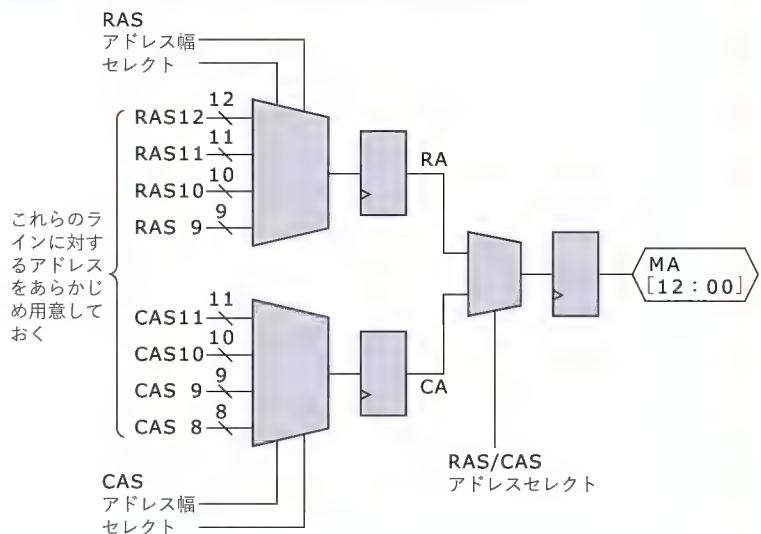
とくに今回は，理解しやすいSDRAMコントローラという意味から，最適化に関しては考慮せずにVHDLソースコードを記述しています．また，採用したFPGAも，スピードグレード的に最高速のデバイスではありません．これを論理合成ツールと配置配線ツールの最適化だけで，100MHzクロックで動作させることはできません．

● SDRAMコントローラの仕様

そこで今回は，64Mバイトまたは128Mバイトの容量のSO-DIMMに限定して設計することにしました．正確には，ちょうど**図4**で示した，1バンク構成の64Mバイト SO-DIMM，または2バンク構成の128Mバイト SO-DIMMとします．

SH-4からは32ビット通常バスモード，また64ビットMPXモードでアクセスされても，いったん前章で解説したローカルバスコントローラがアクセスを受け取り，$\overline{CS}$[6：0](エリアチップセレクト信号)をみて，内部ロジック用にリード/ライト方向やアドレス情報，バイトイネーブル情報を生成しています．

SH-4が出力するアドレス情報はA[25]までの最大64Mバイト領域です．よって，これ以上の容量を扱うには7本あるチップセレクトを使用して，そこから上位アドレスを生成しています．

〔図5〕セレクタ/マルチプレクサ構成による遅延の増加

32ビット通常バスモード用のバスコントローラでは，以上のような方法で生成したPA[28：00]，R/$\overline{W}$信号，ライトイネーブル信号，ならびに32ビットデータバスがSDRAMコントローラに接続され，転送スタートを示すTS(Transfer Start)とメモリアクセス完了を示すTA(Transfer Acknowledge)の信号でデータのやり取りを行います．

64ビットMPXモードのバスコントローラの場合，アドレス，データバス，そしてR/$\overline{W}$までは同じですが，ライトイネーブル信号は，アドレスフェーズ時のアクセスアドレスとサイズから生成します．SH-4が出力するPWBE[7：0]は，MPXモード時は使用できません．SDRAMコントローラ側からみれば，ライトイネーブルはどのような生成のされ方であろうとも関係ありませんが，MPXモードを検討している場合には，前章に戻って，ライトイネーブルの作り方を確認してください．

ここで説明するSDRAMコントローラは，プロセッサ側からは32ビット幅で読み書きが行われ，64ビット幅のSO-DIMMメモリを駆動するように設計しています．**表3**に，SDRAMコントローラの入出力仕様や信号名を示します．

● アドレスエンコード

さまざまな容量のモジュールに対応するためには，このアドレス割り当てが重要なポイントであるということはすでに説明しました．今回はRASが12ビット，CASが9ビット，SDRAM内のバンク構成が4バンクのSDRAMを実装した，モジュールバンク構成が1バンクの64Mバイトと，モジュールバンク構成が2バンクの128MバイトのSO-DIMMを想定しています．

図6に，今回設計するSDRAMコントローラのアドレスエンコードテーブルを示します．SO-DIMMのデータバス幅は64ビットなので，プロセッサからのアドレスバス(以下PA)の下位3ビットは使いません．バイト単位でのアクセスには，後述するDQMB信号を使います．PA[28：3]のうち下位側からCASアドレス，RASアドレス，バンク(BA)アドレス，そして最上位ビットをチップセレクトに割り当てました．

この設計で，筆者の手元にあった1バンク構成の64Mバイト SO-DIMMと2バンク構成の128Mバイト SO-DIMMの両方が，問題なくアクセスできています．

● バイトイネーブル制御

さきのSO-DIMMの構造で説明したように，SO-DIMMは1バイト単位で読み書きを行うことができるように，DQMB信号が8本(ECCやパリティ機能付きであれば9本．本執筆では説明しない)用意されており，DQMBとデータバスDQはおのおの次のようなバイト対応になっています．

DQMB[7]⟷DQ[63：56]

〔表3〕SDRAMコントローラの入出力仕様や信号名

項目		内容
RAS幅		12ビット
CAS幅		9ビット
メモリバンク数		4
CL値		2
t_{RCD}		1クロック(16ns)
t_{RP}		3クロック(48ns)
サポートコマンド	ACTV	バンクアクティブ
	READA	オートプリチャージ付きリード
	WRITEA	オートプリチャージ付きライト
	MSET	モードセット
	REFR	リフレッシュ
バースト長		1ワードアクセスのみ
バースト方式		リードバースト不可/ライトバースト不可

(a) 対応SDRAM仕様

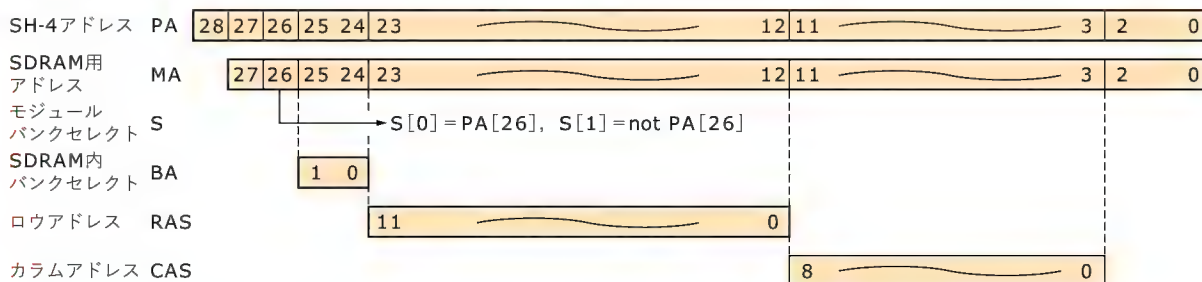
▶ローカルバス側

信号名	内容
CLK	動作クロック
RST	システムリセット
PA[28：00]	プロセッサアドレスライン
PDI[31：00]	プロセッサデータ入力
PDO[31：00]	プロセッサデータ出力
PR/W	リード/ライト転送方向指示
PWBE[3：0]	ライトバイトイネーブル
TS	転送スタート要求
TA	アクセス完了応答

▶SO-DIMM/SDRAM側

信号名	内容
MA[12：00]	RAS/CAS時分割アドレスライン
MDI[63：00]	メモリデータ入力
MDO[63：00]	メモリデータ出力
MCS[1：0]/MRAS/MCAS/MWE	メモリコマンドライン
MDQM[3：0]	メモリデータマスク

(b) 入出力ポート仕様

〔図6〕設計したSDRAMコントローラのアドレスエンコード

DQMB[6]⟷DQ[55：48]
DQMB[5]⟷DQ[47：40]
DQMB[4]⟷DQ[39：32]
DQMB[3]⟷DQ[31：24]
DQMB[2]⟷DQ[23：16]
DQMB[1]⟷DQ[15：08]
DQMB[0]⟷DQ[07：00]

さらに今回，SH-4のローカルバスコントローラを32ビットバス幅で動作させた場合，バイトイネーブル情報は当然ながら32ビット幅分しか出力されてきません．そこで，64ビット幅のメモリを32ビット幅ずつに割り当てなければなりません．ここでは上位側DQ[63：32]と下位側DQ[31：00]の二つのグループに分け，PA[2]が"0"のときにはDQ[63：32]の上位側32ビットを，PA[2]が"1"のときはDQ[31：00]の下位側32ビットを使うよう，ビッグエンディアンのルールを適用することにしています．

メモリライトアクセスのときは，64ビットバスの上位側と下位側32ビットの両方にプロセッサ側のデータPDI[31：00]をセットします．そして，PA[2]が"0"のときにはDQMB[7：4]にPWBE[3：0](プロセッサライトイネーブル信号)をセットして，バイトイネーブル情報を位置が有効であることを示します．PA[2]が"1"のときはDQMB[3：0]は"H"レベルに固定します．

前記SDRAMの動作解説でも紹介したとおり，DQMB信号が"L"レベルであれば，そのバイト位置は書き込み対象位置とされるので，プロセッサからのデータは上位[63：32]データバス中のいずれかのバイト位置のみに書き込まれ，下位[31：00]にはDQMBが"H"であるために書き込みは行われません．

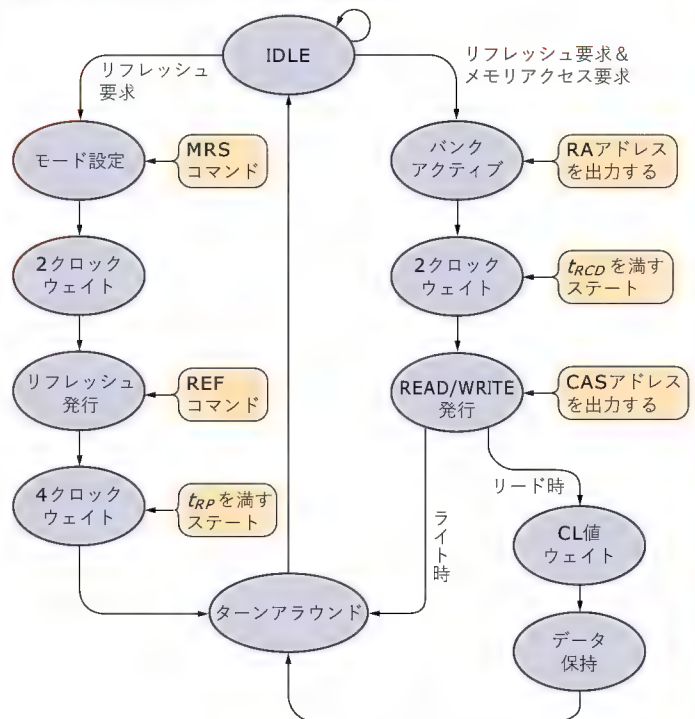
メモリリードサイクル時はDQMB[7：0]のすべてのバイト位置を"L"レベルに設定し，これにより64ビットを一気に読み出しています．そしてPA[2]をみて"0"であればDQ[63：32]をプロセッサ側への読み出しデータ出力としてPDO[31：00](プロセッサ出力データバス)にセットします．同様にPA[2]が"1"であればDQ[31：00]をPDO[31：00]にセットします．

3 SDRAMコントローラのステートマシン

SDRAMコントローラの設計にはステートマシンを使った方法が理解しやすいでしょう．**図7**に設計したSDRAMコントローラの状態遷移図を，**リスト1**(章末)にSDRAMコントローラのVHDLソースを示します．それでは各ステートについて説明します．

● アイドルステート

本ステートでは，プロセッサからのTS信号のアサートによる転送開始要求の待ちと，リフレッシュコントローラからのRefresh_Req信号(リフレッシュサイクル発行要求)の二つを待ちます．

優先順位としてはリフレッシュサイクル発行要求のほうがプロセッサからの転送要求よりも高く，同時に両要求が発行された場合にはリフレッシュが優先されます．この場合はモード設定ステートに遷移します．同時に，リフレッシュコントローラに対して「リフレッシュ要求を受け付けました」という意味で，Refresh_Req信号のネゲートを要求するRefresh_Clr(リフレッシュ要求応答信号)を"H"レベルにアサートします．

〔図7〕SDRAMコントローラの状態遷移図

リフレッシュサイクルが要求されておらず，TS信号は1クロックしか出されないことを想定し〔SH-4の$\overline{BS}$(バスサイクルスタート信号)が1クロックしか発行されないため〕，TS信号と，TAが発行されるまでTSがアサートされたことを示すhold_TS信号のいずれかが"L"レベルであるとバンクアクティブステートに遷移します．

Refresh_ReqやTS/hold_TSがアサートされていない状態では，当然このステートにとどまります．以降のステートマシンの解説でも同様ですが，VHDLの場合は「それ以外の状態のとき」という状態をとくに記述しなくても同一ステート状態が保持されるので，筆者の場合は記述していません．

しかし，Verilog-HDLの場合には，これを書いておかないとステート値がオールゼロの状態にリセットされることになります．この点を注意してソースを読んでください．

● モード設定〜リフレッシュ動作

SDRAMはメモリとしてアクセスを行う前に動作モード設定のコマンドを発行し，CL値やバースト転送長などを設定しなければなりません．

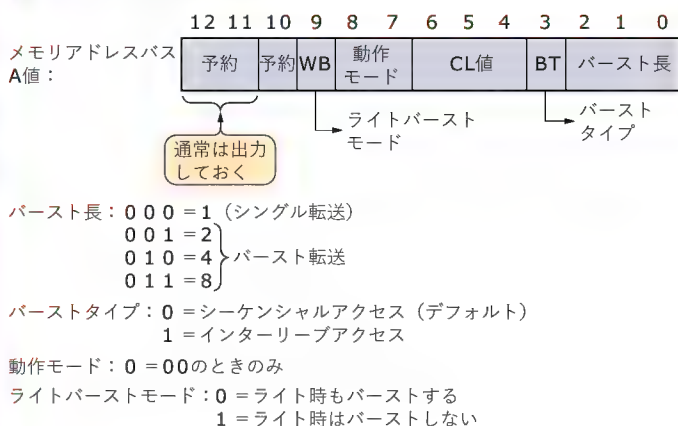
通常この動作は，電源が安定してから規定回数のリフレッシュを行った後に1回だけ行えばよいことなのですが，逆にいえば「規定時間を計算して」，「1回だけコマンド発行をする」という設

〔図8〕モード設定テーブル

バースト長：000＝1（シングル転送）
001＝2
010＝4 ｝バースト転送
011＝8

バーストタイプ：0＝シーケンシャルアクセス（デフォルト）
1＝インターリーブアクセス

動作モード：0＝00のときのみ

ライトバーストモード：0＝ライト時もバーストする
1＝ライト時はバーストしない

計を行うほうが面倒です．そこで，今回のSDRAMコントローラではリフレッシュ動作を行うときにあわせてモードセットを行っています(**図8**)．

Refresh_Reqのアサートによりアイドルステートから本ステートに分岐しましたが，ここでまずMSETコマンドを発行してモード設定を行います．モード設定値はSDRAMに対するアドレスバスにセットされ，$\overline{\text{CS}}$/ $\overline{\text{RAS}}$/$\overline{\text{CAS}}$/$\overline{\text{WE}}$の組み合わせによるMSETコマンドの発行で，SDRAMメモリのクロックの立ち上がりに同期してメモリ側に取り込まれます．

SDRAMコントローラの動作でも説明した，CL値，バースト転送長などは，この操作によりメモリ側に設定します．

アドレスにセットするビット位置に対する値の意味は，JEDECをはじめとして各SDRAMメモリで共通化されていて，データシートにも記載されています．SDRAMであれば，今回紹介した設定値を使っていれば，どのようなメモリが搭載されても動きます．

モード設定ステートの次は1クロックのウェイトを入れた後，リフレッシュコマンドを発行しています．リフレッシュコマンドも，モード設定と同様に単に$\overline{\text{CS}}$/ $\overline{\text{RAS}}$/$\overline{\text{CAS}}$/$\overline{\text{WE}}$を規定の値にしてドライブするだけです．

ただし，リフレッシュコマンド発行後に続けてバンクアクティブ(後述のステートで解説)コマンドを発行するためには，ある一定のインターバルをとらねばなりません．この数値はSDRAMメモリのデータシートで，t_{RCA}(オートリフレッシュ周期)などと記されていて，60ns程度の時間が必要です．この時間が経過しないとコマンドを受け付けることはできないので，本設計ではこのステートで4クロック経過後にアイドルステートに戻るようにしています．同時に，アイドルステートでアサートしておいたRefresh_Clr信号をネゲートして，次のリフレッシュサイクルの開始をリフレッシュカウンタに指示します．

● プリチャージの必要性

余談ですが，SDRAMの場合，リフレッシュサイクルを発行する前には必ずメモリ内部の全バンクをプリチャージしておかなければなりません．これを行わないでリフレッシュを発行すると，リードサイクル時はとくに問題はないようですが，ライトサイクル時のデータはほぼ間違いなくデータが化けるか，書き込み動作がされません．

プリチャージ動作というものが，ライトしたデータを正しくメモリセルに固定する……というような意味であるととらえれば，プリチャージの後にリフレッシュ操作が必要ということも理解できるかと思います．

このプリチャージという操作でライトデータがメモリセルに「定着」すると考えれば，後述の「バンクアクティブ」コマンドに対して「バンククローズ」という言葉を使うと，よりSDRAMの動作が理解しやすいのではないでしょうか．つまり，SDRAMのアクセスは，

バンクアクティブ→リード/ライトアクセス→プリチャージ
(バンククローズ)

というアクセスで一連の動作が完了します．

逆にいえば，プリチャージを行っていないバンクは，リフレッシュサイクルを発行しないかぎり，ずっと使い続けることができるため，SDRAMの性能を引き出すためにはこのテクニックを使います．これは，第4章のグラフィクスボードの設計でのVRAMの使い方で紹介します．

本章における設計では，回路の動作を簡略化してSDRAMを動かすことを前提としているため，プリチャージ機能付きのリード/ライトコマンドの発行をしています．よって別途プリチャージという動作は必要ないので，いきなりリフレッシュを行っているというわけです．

● バンクアクティブ～t_{RCD}ウェイト

アイドルステートからリフレッシュサイクルが要求されておらず，プロセッサからの転送開始要求が行われた状態で，本ステートに遷移されます．

本ステートでは，バンクアクティブコマンドを発行することにより，メモリアクセスを開始します．SDRAMは，リードもしくはライト動作を行う場合には，対象となるメモリバンクがアクティブ状態となっていなければならないので，本ステートが必要となります．

バンクアクティブコマンド発行時には，アドレスエンコードの節で説明したRASアドレスラインやバンクアドレス，チップセレクト$\overline{\text{CS}}$[1：0]をセットし，さらに$\overline{\text{RAS}}$を"L"レベルにアサートします．

このコマンドを1クロックのみアサートしたのち，$\overline{\text{RAS}}$を"H"レベルに戻して2クロックウェイトするステートに移行します．

バンクアクティブ発行から続くREADA/WRITEAコマンド発行までには，t_{RCS}(RAS to CASディレイ)というウェイトを満たすことが規定されています．これを守らないと正しくバンクアクティブが行われず，続くREADA/WRITEAコマンドも有効になりません．そのためこのステートで1クロック(66MHzで16ns)のウェイトを入れているというわけです．

News Flash ― 松下，地上波ディジタル放送用OFDM復調LSIを発売
松下電器産業(株)は，国内の地上波ディジタルテレビ放送と地上波ディジタル音声方式の1セグメント受信が可能な地上波ディジタル放送用OFDM復調LSI「MN88445」を発売した．サンプル価格は¥3,000．

そして1クロック間ウェイトをかけた後に，読み込み/書き出しコマンドを発行するためのステートに遷移します．

● READA/WRITEA発行ステート

前のステートでバンクアクティブされたメモリバンクに対して，次のような動作を行います．まず，CASアドレスを出力し，メモリのアドレスバスにバスコントローラ側からの下位アドレスをセットします．次にバスコントローラ側からのPDI[31：00]（プロセッサ入力データバス）をメモリデータバスMDO[63：32]，ならびにMDO[31：00]にコピーします．ライトサイクル時であればPA[02]を見て，PA[02]=0であれば，DQMB[7：4]に，1であればDMQB[3：0]にPWBE[3：0]の値を設定します．リードサイクル時であればDQMB[7：0]はすべてゼロを設定し，8バイト分すべてを読み出します．そしてデータアクセス方向を示すPR/W信号により，READA（オートプリチャージ機能付きリードサイクル）またはWRITEA（オートプリチャージ機能付きライトサイクル）コマンドを発行します．このコマンドを発行するとき，MA[10]信号を“H”レベルにします．

このとき，$\overline{\text{CS}}$[1：0]とバンクアドレスを示すBA[1：0]は，バンクアクティブ時に発行したときと同一の値を出力しなければなりません．バンクアクティブコマンドでオープンにしたバンクですが，READA/WRITEAコマンド発行時は同一バンクに対する操作になります．$\overline{\text{CS}}$[1：0]とBA[1：0]は，先にオープンしたバンク時に使ったのと同一でなければならないからです．バンクアクティブを発行したから，BA信号は使わなくてもいいや……というわけにはいかず，無視すると別バンクへのアクセスになってしまうので注意してください．

ところで，READA/WRITEAコマンドとREAD/WRITEコマンドの唯一の違いは，$\overline{\text{CAS}}$をアサートする際のMA[10]アドレス信号の状態です．MA[10]が“H”レベルであれば，$\overline{\text{CAS}}$と同時に発行されたREAD/WRITEコマンドはオートプリチャージ機能付きになり，リフレッシュサイクルを発行する際や，RAS/BAアドレスが切り替わったときに，現在オープンになっているバンクを閉じるためのプリチャージコマンドの発行を行わなくてもすみます．

逆にMA[10]が“L”レベルであると，リフレッシュサイクルやバンクをまたぐアクセス時には必ずプリチャージが必要になります．しかし，それ以外のときにはバンクアクティブを行う必要がないために，ビデオ出力で使用するVRAMのような連続したアドレス時には有効です．筆者もSDRAMのアクセス用途によって使い分けています．

最後はステート分岐の記述です．PR/Wを見て，リードサイクルが要求されているのであればCL値で設定されたデータ読み出しタイミングまで待つ，READA処理ステートに移行します．

ライトサイクルのときはこの時点でデータ転送が完了になり，その旨をバスコントローラに通達するためにTA信号をアサートし，同時にSDRAMコントローラとしての最終処理ステートである完了ステートに遷移します．

● READA処理ステート

SDRAMからのデータ読み出し時には，READ/READAコマンド発行をしてからしばらくしないとデータが読み出しできません．この時間はメモリの規格として規定されているCL値で示され，たとえばCL=2の場合には，READ/READAコマンドをメモリが受け取ってから2クロック後にメモリがデータバスをドライブします．よってREADAコマンド発行時には，CL値分だけデータバスの値を内部のレジスタに取り込むのを遅らさなければなりません．

このステートでは，WCNTカウンタを用意しておき，このカウンタの値によってCL値にあわせたクロック数をカウントしています．

ステートマシンへ分岐後，まずWCNTをカウントアップします．FPGA内部はメモリへの制御信号ドライブより1クロック先に動作が進んでいるので，CL値が2であればここで3クロック待ちます．もしCL値が3であればこのステートで4クロック待つことになります．

この値だけ待ったときのデータバス上のデータをバスコントローラに返す（PDO[31：00]にセットする）ことで，正しくメモリデータを取り込めるようになります．

なお，データをセットする際にはDQMB信号生成時と同様な注意が必要です．つまり，メモリから読み出されるデータは64ビット長のデータですが，バスコントローラにセットするデータは32ビット長です．よって，ここでもPA[2]をみて，“0”であればメモリデータバスの上位4バイト，MDI[63：32]をPDO[31：00]にセットし，PA[2]が“1”であればメモリデータバスの下位4バイト，MDI[31：00]をPDO[31：00]にセットします．

次に，データをセットしたタイミングから1クロック遅れで，バスコントローラにデータ転送が完了したという旨を通知するためにTA信号をアサートします．この完了通知をバスコントローラが識別して，プロセッサ側に読み出したメモリデータを応答することになります．

ここで，読み出したデータをセットしたタイミングと同時に完了通知をアサートしたほうが1クロック早くなり，パフォーマンスが向上しますが，今回は理解しやすいステートということで，段階を分けて記述/設計しています．

このステートでTAのアサートを行ったと同時に，ライトサイクルと同様にSDRAMコントローラとしての最終処理ステートである完了ステートに遷移します．

● 完了ステート

SDRAMコントローラで実際にメモリのリード/ライトにかかわるステートではいちばん最後に到達するステートです．

このステートはそれまでセットした各信号のリセットを行い，アイドルステートに戻る準備を行います．

- バスコントローラへのアクセス応答であるTA信号をネゲートする
- メモリアドレス/バンクアドレスをゼロに戻す

- $\overline{CS}$/ $\overline{RAS}$/$\overline{CAS}$/$\overline{WE}$のすべてを“H”レベルに戻し，NOP コマンドを発行する
- DQMB ：データマスク信号を“H”レベルに戻し，不慮のライトを行わないようにする
- READA 発行後のステートで使用していた WCNT（ウェイトカウンタ）をオールゼロにクリアする
- SDRAM のデータバスドライブ用出力イネーブル信号をネゲートし，SDRAM のデータバスをハイインピーダンス状態にする

つまり，電源投入直後の何もアクセスされていない状況に戻す処理を行うわけです．

じつは実際には不要な処理もあります．たとえば，個々のステートで $\overline{CS}$/ $\overline{RAS}$/$\overline{CAS}$/$\overline{WE}$や DQMB 信号はすでにネゲートされているはずですし，MA や BA はアクセス直後の状態を出力し続けていてもかまいません．

しかし，この完了ステートの役割はもう一つあり，ステートマシンの状態が何かの拍子に狂ってしまい，ここで紹介したステート状態以外の値になったとき，とにかくにもこの完了ステートからアイドルステートへ復帰させたいときにも使う場合があるのです．

通常の運用状態ではありえないことなのですが，たとえばクロックサイクルが急峻に変動した場合や電源電圧が一時的に下がってステートマシンが異常状態に陥ったとき，ステートマシンの最後に，When others 文により記述を行うことで，この完了ステートで各信号を初期化するように指示しています．この構文は「これ以外のステート状態に陥ったらターンアラウンド（完了ステート）に飛びなさい」というものです．

実際筆者の場合，クロックアップを行ってステートマシンの耐性を試験する場合がありますが，ステートの読み飛ばしでこの Others 構文が実行される場合があることを確認したことがあります．このような処理構文を使って完了ステートを利用するという側面もあるため，リセット時と同様な処理を行っているのです．

〔図 9〕バスのアクセス波形

(a) バイトアクセス時

(b) ロングワード(4バイト)アクセス時

4 リフレッシュコントローラ

最後に，DRAM 制御には必要不可欠なリフレッシュコントローラについて説明します．リフレッシュコントローラといっても，実際にリフレッシュ動作を制御するのは SDRAM コントローラのステートマシンに入っているので，正確にはリフレッシュ要求を行うカウンタ群とリクエスト信号を発行するフリップフロップをまとめた，リフレッシュタイミングコントローラです．

回路は単なるアップカウンタで，クロックを数えておおよそ 15.6 μs を数えると Refresh_Req 信号を“H”レベルにアサートします．Refresh_Req 信号は，ステートマシンがリフレッシュサイクルの要求を認識するまでアサートしつづけ，Refresh_Clr 信号の“H”アサートにより，Refresh_Req 信号とカウンタのリセットを行っています．

本来ならば，SPD に書かれたリフレッシュインターバル情報からリフレッシュ周期を算出して，動的に変更できるアップカウンタ（ダウンカウンタでも何でもよい）を組むのが正しいのでしょうが，15.6 μs より短い時間を要求する SDRAM を筆者は見たことはないので，決め打ちで設計しました．

余談ですが，DDR メモリでは 7.8 μs/1 リフレッシュ周期が一般的のようで，動作周波数によりカウンタ値は変えていかなければならないため，リフレッシュコントローラ内のカウンタはプリセット機能付きにして，CPU からリフレッシュ周期を可変できるほうが実用的だと思います．

5 設計した SDRAM コントローラの動作

ここで設計した SDRAM コントローラを，第 1 章で

紹介した64ビット版ローカルバスコントローラと組み合わせて，128Mバイト SO-DIMMでテストした結果の波形を示します．

● **バイト(8ビット)アクセス時の波形**

図9(a)はアドレス0000_0001hに1バイトのライトとリードを行った場合のバスの波形です．ライトサイクルではWRITEAコマンド発行時，PA[2]が"L"レベルなので，SH-4のPWBE[3：0]信号を64ビット幅SDRAMのDQMB[7：0]のうち下位側に出力し，上位側はすべて"H"レベルにします．すると結果的に，DQMB[7：0]のうち実際に書き込み動作をするDQMB[1]の1バイト分のみが"L"レベルになっていることがわかります．デバイスセレクトとして動作する$\overline{CS}$0信号は"L"なので，SDRAMチップそのものはすべてアクティブ状態になるのですが，DQMB[1]だけがアサートされるので，実際に書き込まれるのは1バイト分のみになります．

リードサイクルでは，DQMBを全ビット"L"にして64ビット幅で読み出しているため，SDRAMデータバスはREADAコマンド発行後，CL＝2経過ののちにデータバスをドライブしています．

● **ロングワード(32ビット)アクセス時の波形**

図9(b)はアドレス0000_0004hにロングワード(32ビット)のライトとリードを行った場合のバスの波形です．今度はPA[2]が"H"レベルなので，SH-4のPWBE[3：0]信号をDQMB[7：4]に出力し，下位側はすべて"H"レベルにします．これにより64ビットのSDRAMのうち上位4バイトにデータが書き込まれます．

まとめ

以上で，SDRAMに対する基礎知識，SDRAMを応用したSO-DIMMの特徴とSPDの解説，ならびにSDRAMコントローラの設計方法を解説しました．

SDRAMの知識さえあれば，パソコンでは現在主流のDDRメモリや，その次の主流となるであろうDDR-IIメモリ，また1T-SRAMやMCRAMでも動作基本を理解することは容易になると思います．

組み込みマイコンのほとんどがSDRAMを自身でサポートしているために，このあたりは意識して設計する機会が失われていますが，ハイエンドプロセッサをどうしても使いたい場面ではチップセットや内蔵メモリコントローラの性能では仕様を満足できず，自身でカスタマイズしなければならない場面も出てくることでしょう．

さらにFPGAであればこその利点で，実際初期のSH-4(7750/7750S)では256MビットのSDRAMはサポートされていない(7750Rは接続可)のですが，FPGAでホストブリッジを構成しておけば，このあたりは柔軟に対応できます．

参考文献

1) 井倉将実，「CPLDによるSDRAMコントローラとSDRAMメモリボードの製作(前後編)」，『Interface』，1999年8月号/10月号
2) 井倉将実，「SDRAM DIMM搭載PCIメモリボードの設計/製作(前後編)」，『Interface』，2000年2月号/3月号

いくら・まさみ　来栖川電工有限会社

〔**リスト1**〕**SDRAMコントローラのVHDLソース**(一部)

```
SDRAMC_Ctrl   : process ( BUSCLK, nRESET )

    variable  SDRAMC_Current_State    : SDRAMC_Sequensor ;     --:= SDRAMC_IDLE ;
    variable  SDRAMC_Next_State       : SDRAMC_Sequensor ;     --:= SDRAMC_IDLE ;
    variable  WCNT                    : std_logic_vector(2 downto 0) := "000" ;

begin

    if ( nRESET = '0' ) then

    -- SH7750 Bus I/F
         SDRAMC_Current_State     := SDRAMC_IDLE ;
         SDRAMC_Next_State        := SDRAMC_IDLE ;

    --
         MA              <= ( others => '0' ) ;
         MBA             <= "00" ;

         nMCS            <= '1' ;                          -- NOPコマンド発行状態にする
         nMRAS           <= '1' ;
         nMCAS           <= '1' ;
         nMWE            <= '1' ;
         DQM             <= ( others => '1' ) ;

         MD_O            <= ( others => '0' ) ;
         MD_O_OEN        <= '0' ;
         Refresh_Clr     <= '0' ;
         WCNT            := "000" ;

    elsif ( BUSCLK'event and BUSCLK = '1' ) then

         MD_O(63 downto 32) <= SDR_WrDB ;                  -- 32bit CPUデータ出力
         MD_O(31 downto  0) <= SDR_WrDB ;                  -- 32bit CPUデータ出力

    -- ********************** --
         u1 : for i in 0 to 15 loop                        -- 64bit SDRAMデータ出力
```

〔リスト1〕SDRAMコントローラのVHDLソース(一部)(つづき)

```
            cpy SDR A2(i)  <= SDR A(2) ;
            if ( pHld MemData(i) = '1' ) then
                if ( cpy SDR A2(i) = '0' ) then              -- A(2)をみて,64ビット中から選択する
                -- Capture LOWER data.
                    SDR RdDB(i*2+1 downto i*2)   <= MD I(i*2+1  downto i*2   ) ;
                else
                -- Capture UPPER data.
                    SDR RdDB(i*2+1 downto i*2)   <= MD I(i*2+33 downto i*2+32) ;
                end if ;
            end if ;
        end loop;

    -- **** ステートマシン定義 **** --
        SDRAMC Current State    := SDRAMC Next State ;

        case SDRAMC Current State is

        -- ******** --
            when SDRAMC IDLE =>
                if ( Refresh_Req = '1' ) then                -- リフレッシュサイクルだったら
                    Refresh Clr   <= '1' ;                   -- リフレッシュ要求クリア
                    SDRAMC Next_State := SDRAMC_ModeSet ;
                elsif ( nTS = '0' or hold nTS = '0' ) then   -- リフレッシュがこなく,MPUアクセスだったら
                    if ( SDR RD = '0' ) then                 -- (ライトサイクル時は...)
                        MD O OEN  <= '1' ;                   MDバス出力開始
                    end if ;
                    SDRAMC Next State := SDRAMC SH RAS Assert ;
                end if ;

        -- ******** SDRAMモード設定ステート ********* --
            when SDRAMC ModeSet =>
                -- モード設定コマンドの発行
                MA       <= SDRAM Mode ;                     -- アドレスバスにコマンドを設定.
                MBA      <= "00" ;
                nMCS <= '0' ; nMRAS <= '0' ; nMCAS <= '0' ; nMWE        <= '0' ;
                DQM      <= ( others => '1' ) ;              -- DQMはディゼーブルのこと!!

                SDRAMC Next State := SDRAMC Wait MSET2REFR1 ;

            when SDRAMC Wait MSET2REFR1 =>
                SDRAMC Next State := SDRAMC Wait MSET2REFR2 ;  -- 2クロック待ち
            when SDRAMC_Wait_MSET2REFR2 =>
                SDRAMC Next State := SDRAMC REFR ;

        -- ******** リフレッシュコマンド発行ステート ******** --
            when SDRAMC_REFR =>
            -- リフレッシュコマンドの発行
                nMCS <= '0' ;  nMRAS     <= '0' ;  nMCAS     <= '0' ;     nMWE      <= '1' ;

                SDRAMC Next State := SDRAMC Wait REFR2IDLE ;

        -- ******** リフレッシュ終了処理ステート ******** --
            when SDRAMC Wait REFR2IDLE =>
                nMCS <= '1' ;  nMRAS     <= '1' ;  nMCAS     <= '1' ;     nMWE      <= '1' ;

                Refresh Clr    <= '0' ;                      -- リフレッシュ要求クリア信号のリセット

                WCNT  := WCNT + 1 ;                          -- 4クロック待ってから
                if ( WCNT = "100" ) then                     -- 完了ステートに移行する
                    SDRAMC Next State := SDRAMC SH Turn around ;
                end if ;

        -- ******** バンクアクティブ操作 ******** --
            when SDRAMC SH RAS Assert =>
                MA       <= SH RAS Adrs ;                    -- アクティブバンクコマンドを発行
                MBA      <= SH BANK Adrs ;
                nMCS <= '0' ; nMRAS <= '0' ; nMCAS <= '1' ; nMWE        <= '1' ;

                SDRAMC_Next_State := SDRAMC SH Wait tRCD ;

        -- ******** tRCDパラメータ待ちステート ******** --
            when SDRAMC_SH_Wait_tRCD =>
                MA       <= SH CAS Adrs ;                    -- CASアドレス出力開始
                MBA      <= SH BANK Adrs ;
                SDRAMC Next State := SDRAMC SH Wait tRCD2 ;

            when SDRAMC SH Wait tRCD2 =>                     -- 2クロック待つ
                SDRAMC Next State := SDRAMC SH CAS Assert ;

        -- ******** --
            when SDRAMC SH CAS Assert =>
```

〜以下省略〜

第3章

入出力機能拡張はすべてPCIバス上に実装する

PCIホストコントローラの設計/製作

井倉將実

今回設計するシステムでは，各種I/Oインターフェースをすべて PCI バス上に配置するアーキテクチャを採用した．画面表示をするにも，キーボード入力をするにも，すべてPCIバスを経由したアクセスとなる．ここではPCIバスの基本的な動作と，システムにPCIバスを実装する場合に必須となるホスト機能について解説し，PCIホストコントローラを設計/製作する．
（編集部）

はじめに

今回のシステムでは，CPUやメモリを載せたプロセッサボードは，それ単体ではI/O機能をもっていません(写真ではコンパクトフラッシュスロットやIDEコネクタが見えるが，これは別の目的で使用するために用意したもので今回は使えない)．画面出力やキーボード入力は，すべてPCIバス経由で行うアーキテクチャを採用したからです．つまり，PCIホストコントローラがなければ手も足も出ないわけで，本システムの拡張性を左右するもっとも重要な部分です．

まず，PCIバスに関連した用語について説明し，今回設計したPCIホストコントローラの仕様，そしてイニシエータシーケンサについて解説します．誌面に余裕がないので，PCIバスにおける一般的な用語については簡潔に説明します．参考資料として，本誌増刊TECH I Vol.3『PCIデバイス設計入門』〔参考文献1)〕を参照してください．

1 PCIバスの概要

1.1 バス構成と信号

● **PCIバスを搭載したシステムの構成例**

図1に，PCIバスを搭載したシステムの構成例を示します．バスアクセスを発生させる主体となるデバイスのことをバスマスタと呼びます．今回設計するPCIホストコントローラとは，CPUからのアクセスを受けてPCIバス上でバスマスタとして動作する，CPUローカルバスとPCIバスをブリッジするデバイスであるともいえます．

PCIバスでは複数のバスマスタデバイスが存在できるので，マルチバスマスタシステムとも呼ばれます．しかし一つのバス上で複数のバスマスタが同時に動くことはできません．交通整理をするように，適時バスの制御権を与えていくことをバスアービトレーションと呼び，それを行うバスアービタはPCIバスシステムに必ず実装されます．

バスの制御権を取得して動作を開始したデバイスをイニシエータと呼びます．ある瞬間には一つのイニシエータと一つのターゲットという1対1の間(エージェント間)での転送しか行えません(一部のアクセスを除く)．

● **PCIバスの信号概要**

次に，PCIバスの各信号の意味を説明します．

▶ CLK(クロック)

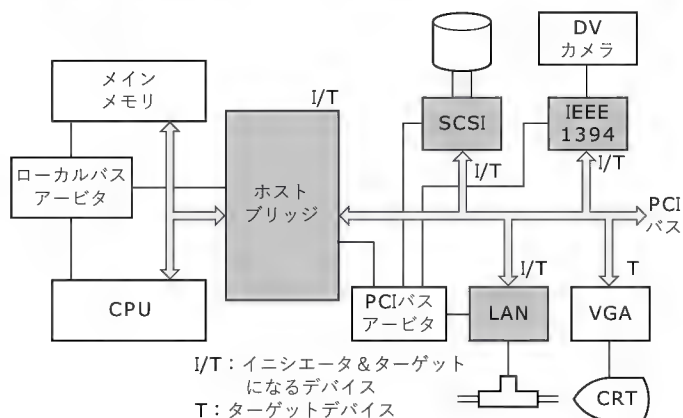
PCIバスの動作の基準となるクロック信号です．リセットと割り込み信号以外のすべての信号は，この信号を基準として動作します．

▶ RST#(リセット)

システムの電源投入時またはシステムリセット時にアサートされる，PCIバスのリセット信号です．PCIバスクロックとは同期していません．

▶ AD[31：0](アドレス/データバス)

データバスとアドレスバスは時分割でこの32本の信号線を使います．アドレスフェーズではアドレスが，データフェーズでは

〔図1〕PCIバスを搭載したシステムの構成例

〔表1〕PCIバスコマンド

バスの動作			C/BE〔3：0〕#
メモリサイクル	リード	メモリリード	“L”“H”“H”“L”
		メモリリードライン	“H”“H”“H”“L”
		メモリリードマルチプル	“H”“H”“L”“L”
	ライト	メモリライト	“L”“H”“H”“H”
		メモリライトアンドインバリデート	“H”“H”“H”“H”
I/Oサイクル	I/Oリード		“L”“L”“H”“L”
	I/Oライト		“L”“L”“H”“H”
コンフィグレーションサイクル	コンフィグレーションリード		“H”“L”“H”“L”
	コンフィグレーションライト		“H”“L”“H”“H”
インタラプトアクノリッジサイクル			“L”“L”“L”“L”
スペシャルサイクル			“L”“L”“L”“H”
デュアルアドレスサイクル			“H”“H”“L”“H”
未定義，予約コマンド			“L”“H”“L”“L”
			“L”“H”“L”“H”
			“H”“L”“L”“L”
			“H”“L”“L”“H”

データが出力されます．

▶ C/BE[3：0]#（バスコマンド/バイトイネーブル）

この信号線も時分割で使われていて，アドレスフェーズでは後述するバスコマンドが出力され，データフェーズではバイトイネーブルとして動作します．

▶ FRAME#（フレーム）

イニシエータがアクセスを開始するときにアサートします．またバースト転送中はアサートされ続けます．

IRDY#（イニシエータレディ）

イニシエータがデータ転送可能状態にあるときにアサートします．

▶ DEVSEL#（デバイスセレクション）

アクセスを受けたターゲットがアサートし，アクセスが終了するまでアサートし続けます．

▶ TRDY#（ターゲットレディ）

ターゲットがデータ転送可能状態にあるときアサートします．よってIRDY#とTRDY#の両方がアサートされているときにデータ転送が行われます．

▶ PAR（パリティ）

PCIバスにはバスパリティがあります．AD[31：0]とC/BE[3：0]#の合計36本のうち，立っているビットが偶数なら“L”，奇数なら“H”を出力します．

▶ STOP#（ストップ）

ターゲットがイニシエータに対して，アクセスを中断してもらうときにアサートする信号です．DEVSEL#やTRDY#のディアサートと組み合わせることで，アクセスの中断方法にいくつかの種類があります．

IDSEL（PCIデバイスセレクト）

コンフィグレーションサイクルのときのみ意味をもつ信号で，コンフィグレーションサイクルのアドレスフェーズ時にこの信号が“H”だった場合は，自分が選択されていることを示します．

▶ INTA#～INTD#（インタラプトA～D）

PCIデバイスからの割り込み信号です．PCIの割り込みは共有可能なので，同じ信号線をいくつかのPCIデバイスで共有する場合もあります．

▶ PERR#（パリティエラー）

リードサイクルではデータを読み取るイニシエータが，ライトサイクルではデータが書き込まれるターゲットが，データフェーズでパリティをチェックし，パリティエラーが発生したときアサートします．

▶ SERR#（システムエラー）

システムにとって致命的なエラーが発生したときにアサートされます．一般的には，アドレスフェーズでパリティエラーが発生したことを検出したらアサートします．

▶ REQ#（リクエスト）

マスタデバイスがバスの制御権をシステム（バスアービタ）に要求するときにアサートします．

▶ GNT#（グラント）

REQ#を要求しているデバイスに対し，システムがバスの使用を許可したときにアサートされます．これがアサートされて，はじめてマスタはイニシエータとしてアクセスを開始できます．

これ以外にもいくつか信号がありますが，ほとんどの場合，使用することのない信号なので，ここでは説明を省略します．

● **PCIバスコマンド**

ISAバスなどでは，MEMRやIOWRといった制御線で，メモリリードやI/Oライトなどバスの動作を示していました．PCIバスでは**表1**に示すようなPCIバスコマンドで，アクセス対象の空間や方向などを示します．

1.2 バスアクセスの開始

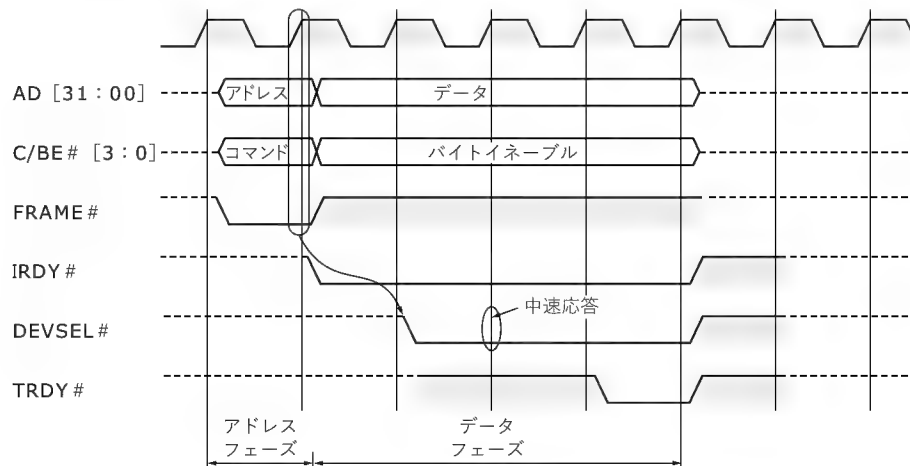
次にイニシエータの立場で，バスの制御権要求からバスアクセス（トランザクションと呼ぶ）の開始，そして終了までを見てみます．とくに各種エラー発生時にどのように処理するのかを重点的に解説します．

● **アービトレーションフェーズ**

バスの制御権要求方法については後で解説します．ここではバスの制御権を取得できたとします．しかしバスの制御権を取得できたからといって，ただちにトランザクションを開始できるわけではありません．まだバス上では他のデバイスによるトランザクションが行われている最中かもしれません．バスがアイドル状態であるかどうかを確認する必要があります．

バスアイドル状態とは，FRAME#が“H”，IRDY#が“H”のときです．この両信号がディアサート状態であれば，バスは誰も使っていないということを示しています．

ターゲットデバイスはDEVSEL#，TRDY#，STOP#を制御しますが，これらの信号はイニシエータデバイスがトランザク

〔図2〕トランザクションの例

ションを起こすことで動きはじめる信号なので，FRAME# や IRDY# が“H”であるならその状態は変化しません．

● アドレスフェーズ

バスがアイドル状態であることを確認したら，アドレスフェーズを開始します．AD バスにアクセス先のアドレスを，C/BE# にバスコマンドを出力し，FRAME# をアサートします．発生させるトランザクションがコンフィグレーションサイクルであれば，特定の PCI デバイスにつながっている IDSEL 信号を“H”レベルにアサートします(詳細は後述)．

アドレスフェーズは FRAME# をアサートした最初の 1 クロックのみです．その次のクロックからはデータフェーズに入ります．データフェーズでは，リードサイクルであればターゲットからのデータ受け入れ準備のために AD バスのドライブ方向を入力に切り替え，データ入力の準備が整ったことを IRDY# をアサートしてターゲットに知らせます．また，ライトサイクルであればターゲットへ書き込むデータを AD バスに出力し，またそのデータがそろったことを示すために IRDY# をアサートします．

またこのとき，そのトランザクションをシングル転送で行う場合は，IRDY# のアサートと同時に FRAME# をディアサートします．バースト転送で行いたい場合は，FRAME# もアサートしたままです．

このように，アドレスフェーズからデータフェーズへの移行，またイニシエータレディを示す IRDY# のアサートなどは，ターゲットの応答とは無関係に進めてかまいません．

図2 がシングル転送のときの一般的なトランザクション例です．図からわかるように，FRAME# が 1 クロック期間のみアサートされた後，FRAME# のディアサートと同時に IRDY# をアサートしてターゲットの応答を待っています．

1.3 データ転送

● データフェーズ

ターゲットが DEVSEL 応答を返し，IRDY# と TRDY# がともにアサートされたクロックでデータ転送が成立します．シングル転送の場合は，IRDY# をアサートするのと同時に FRAME# をディアサートします．データ転送完了後でトランザクションの終了となります．

バースト転送の場合は FRAME# もアサートし続けます．最後のデータを転送するときに FRAME# をディアサートします．データ転送完了後でトランザクションの終了となります．

DEVSEL 応答を返してくるデバイスがいなかった場合は，データ転送が行われずにトランザクションの終了に入ります．さらに，データ転送途中でエラーが発生する場合もあります．エラーの内容により，適切なトランザクション終了処理が必要です．

1.4 バスアクセス終了

イニシエータロジックは，バスアービトレーションフェーズ，アドレスフェーズ，そしてデータフェーズの三つのフェーズで一連のトランザクションを行います．

トランザクションの終了は，トランザクションが正常に終了した場合とエラーが発生した場合に分けられます．さらにエラーが発生した場合には，マスタ側にその要因がある場合と，ターゲット側に要因がある場合に分けられます．

正常終了でもエラー終了でも，エラー時にエラーの原因がどちらにあったにしても，トランザクションを正しく終了(ターミネーション)させるのはイニシエータ側の責任です．

● ターミネーション時の信号制御方法

FRAME# や IRDY# など制御信号の操作には，いくつかの決まりがあります．FRAME# と IRDY# を同時にアサートしたり，同時にディアサートすることはプロトコル違反になります．

すでに説明したような各種エラーを検出したからといって，FRAME# と IRDY# を同時にディアサートしてトランザクションを終了させることはできません．場合によってはダミーサイクルとして，転送データがないにもかかわらず，IRDY# をアサートしなければならないこともあります．

トランザクションを終了する場合の FRAME# や IRDY# の制御の基本ルールとしては，次のようになります．まず，FRAME#

もIRDY#もアサートしていた場合は，先にFRAME#をディアサートし，次のクロックでIRDY#をディアサートします．FRAME#をアサートし，IRDY#をディアサートしていた場合は，そのままFRAME#をディアサートしてしまうと，いきなりバスアイドル状態を示してしまうので，FRAME#をディアサートすると同時にIRDY#をアサートし，1クロック後にIRDY#もディアサートします．

そして，ディアサートした次のクロックで，各信号のドライブを開放します．ADバスやC/BE#は，IRDY#のドライブ開放と同時に信号のドライブを終了します．

● **マスタイニシエーテッドターミネーション**

イニシエータ側の都合でバスサイクルを終了させる場合です．イニシエータ側で何らかのエラーが発生した場合，またエラーなくトランザクションが完了した場合の正常終了時もこれに含まれます．

▶コンプリート

一連のバスサイクルが正常に終了した場合です．シングル転送時はすでにFRAME#がディアサートされているので，IRDY#をディアサートして完了です．バースト転送時でも，最後のデータを転送する場合はFRAME#をディアサートしているので，IRDY#をディアサートして完了です．

▶タイムアウトとプリエンプション

イニシエータは，バスアービタからバス制御権を得てバスサイクルを実行しています．しかし無制限にバスを占有してデータ転送を行えるわけではありません．もっと長時間バスを使いたい場合であっても，ほかにバスを要求しているデバイスが存在する場合は，そのデバイスにもバス制御権を与えなければなりません．

バスアービタの目的は，PCIバス上の各デバイスがデータ転送できるように，バスの制御権を各デバイスに分け与えることです．よって，複数のデバイスからバス制御権が要求されている場合は，バスアービタは現在バスを使用中のイニシエータに対してバスの制御件を与える信号をディアサートし，「ほかのデバイスがバスの制御権を要求しています．なるべく早くバスサイクルを完了してバスを開放してください」と伝えます．

バスの制御権を失ったデバイスは，できるだけすみやかにバスを開放しなければなりません．

▶マスタアボート

トランザクションを開始しても，該当するアドレスやバスサイクルに応答するターゲットがなかった場合の処理です．

イニシエータはトランザクションを開始しすると同時に，イニシエータ内に組み込まれているタイマを起動します．そしてPC/AT互換機の場合には，4クロック以内にDEVSEL#応答がない場合，どのターゲットからも応答がなかったと判定してバスサイクルを終了します(**図3**)．

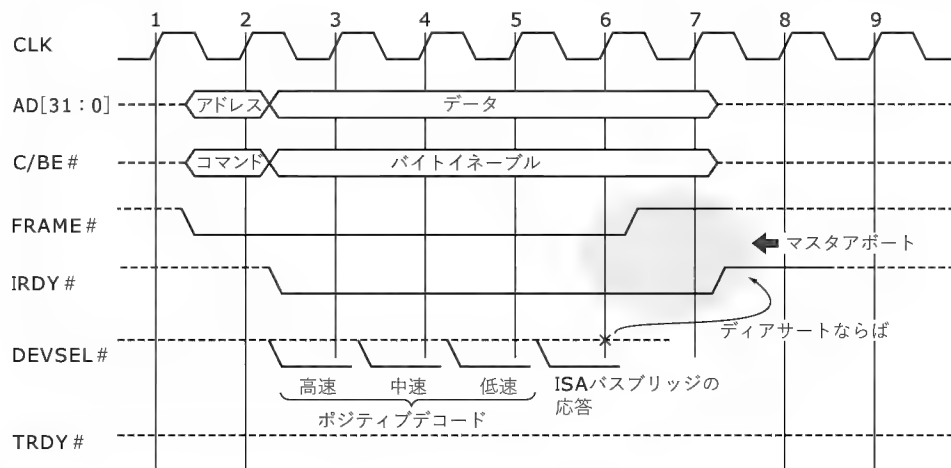
より信頼性の高いバスシステムを構築する場合は，ここでバスエラーなどを発生させるのでしょうが，PCIバスの規格ではそこまでの規定はなく，コンフィグレーションレジスタのステータスレジスタにあるマスタアボート通知フラグに1をセットするとだけ規定されています．

マスタアボートにより終了した場合，ライトサイクルの場合は出力したデータを破棄すればよいのでしょうが，リードサイクルの場合は何もデータを返さないという処理でよいのでしょうか．PCIバスはPC/AT互換機から発展してきたという歴史的経緯からか，デバイスが実装されていない領域を読み出した場合には，FFhを返すようにするのが一般的です．

たとえばコンフィグレーションサイクルでは，存在しないデバイスへのアクセスを行うことも考えられます．システム起動時にコンフィグレーションサイクルを発行し，どのようなデバイスが接続されているを検索する場合，すべてのコンフィグレーション空間に対してリードアクセスを行います．そこにデバイスが接続されていれば何らかのレジスタの値(たとえばベンダIDやデバイスID)が読めるでしょうが，デバイスが未接続の場合は当然マスタアボートが発生します．CPUはベンダIDを読み出したとき，それがFFFFhだった場合は，そこにはデバイスは接続されていないと判断します．

● **ターゲットイニシエーテッドターミネーション**

ターゲットデバイス側の都合でバスサイクルを終了させる場

〔図3〕マスタアボート

〔表2〕ターゲットイニシエーテッドターミネーションの種類

転送モード	データフェーズ	DEVSEL#	TRDY#	STOP#	状態説明
		"H" "H" "H"	"H" "L" "L"	"H" "L" "H"	ターゲットアイドル状態 (ターゲットの動作としてありえない) (ターゲットの動作としてありえない)
シングル転送	1回目	"L"	"H"	"H"	ターゲットウェイト状態
シングル転送	1回目	"L"	"L"	"H"	1ワード転送完了．トランザクション正常終了
シングル転送	1回目	"L"	"L"	"L"	1ワード転送完了．トランザクション正常終了(実質的にディスコネクトとはみなさない)
シングル転送	1回目	"L"	"H"	"L"	リトライ．データ未転送．いったんトランザクションを終了し，再度バス制御権を取得して，同じアドレスでトランザクション開始
シングル転送	1回目	"H"	"H"	"L"	ターゲットアボート．データ未転送．トランザクション終了 同じアドレスでトランザクションは開始しない
バースト転送	n回目	"L"	"H"	"H"	ターゲットウェイト状態
バースト転送	n回目	"L"	"L"	"H"	1ワード転送完了．バースト転送継続
バースト転送	n回目	"L"	"L"	"L"	ディスコネクト．1ワード転送完了．バースト転送打ち切り．いったんトランザクションを終了し，再度バス制御権を取得して，次のアドレスからトランザクション開始
バースト転送	1回目	"L"	"H"	"L"	リトライ．データ未転送．バースト転送打ち切り．いったんトランザクションを終了し，再度バス制御権を取得して，同じアドレスからトランザクション開始
バースト転送	2回目以降	"L"	"H"	"L"	ディスコネクト．データ未転送．バースト転送打ち切り．いったんトランザクションを終了し，再度バス制御権を取得して，同じアドレスからトランザクション開始
バースト転送	n回目	"H"	"H"	"L"	ターゲットアボート．データ未転送．バースト転送打ち切り．いったんトランザクションを終了し，再度バス制御権を取得して，次のアドレスからトランザクション開始 (ターゲットアボード発生アドレスがバースト転送終了アドレスならトランザクション終了)

合です．ターゲットデバイスはSTOP#をアサートして，イニシエータに対してバスサイクルの終了を要求します．

ターゲットイニシエーテッドターミネーションの種類とそのときの各信号の状態を**表2**に示します．ターゲットが制御する信号はDEVSEL#，TRDY#，STOP#の三つです．よって八つの状態が考えられますが，すべてディセーブル状態はバスアイドル状態，また規格上ありえない組み合わせもあるので，組み合わせの数は絞られます．

シングル転送とバースト転送，またバースト転送でも最初のデータフェーズとそれ以降のデータフェーズでは，同じ信号状態でも意味合いの異なるものがあるので注意してください．

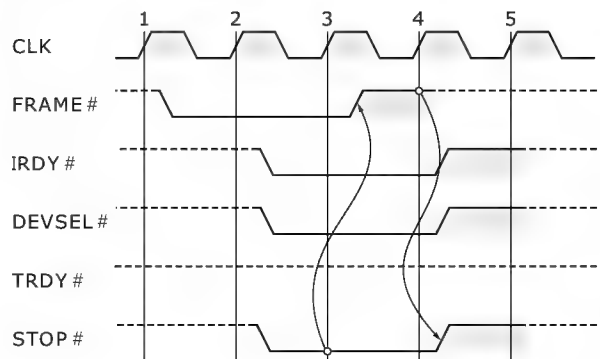
▶リトライ

ターゲットが何らかの理由で，トランザクションに対して一時的に対応できない場合に要求される処理です(**図4**)．リトライを検出したイニシエータは，いったんトランザクションを終了させ，PCIバスクロックで2クロック以上の間を置いてから，再度バスの制御権を要求して，同じアドレスからトランザクションを再開します．

リトライサイクルは，ターゲットデバイスが正常なバスサイクルを受けつけた場合であっても，すぐにはデータ転送応答ができない場合に発行される，通常のターミネーション方法です．よって，イニシエータを設計する場合は，正常終了に対する応答と同じく必須の機能です．

▶ディスコネクト

何度かデータ転送が完了した後に，何らかの理由によりデータ転送を継続できないとき，後でトランザクションを再開してほしい場合に要求される処理です．TRDY#がディアサートで

〔図4〕リトライのタイミング

DEVSEL#とSTOP#がアサート状態ではリトライと同じですが，そのトランザクション内ですでに何度かデータフェーズが成立した後には，ディスコネクトになります．一般的にはバースト転送を途中で打ち切る場合によく使われます．

よって，シングル転送にディスコネクトはありません．ディスコネクトに似たようなタイミングは考えられますが，1ワードも転送しないうちにDEVSEL#とSTOP#がアサートされた場合はリトライですし，DEVSEL#，IRDY#，STOP#がすべてアサートされた場合は，TRDY#がアサートされているので1ワード分のデータ転送が完了し，シングルデータ転送が成立します．この場合のSTOP#のアサートは無視される形になります．

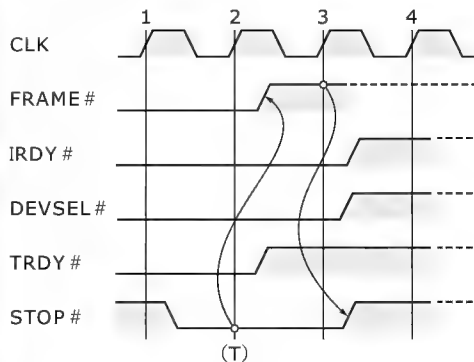
ディスコネクトには，同時にデータ転送が成立する場合とデータ転送をしない場合があります．データ転送をしない場合はそのままターミネーションに入ればよいのですが，データ転送が成立する場合は，そのデータを正しく処理しなければなりません．

図5にディスコネクト例を示します．データ転送の成立と同時にディスコネクトが要求されている例です．クロック2のタイミングでデータ転送を行います．また，ディスコネクトを検出したので，FRAME# を先にディアサートします．そして，次のクロック3でIRDY# をアサートします．クロック3の時点ではデータ転送は行いません．

▶ターゲットアボート

ターゲットアボートとは，「アクセスされた空間はたしかに自分の応答すべき空間ではあるが，そのアドレスにはアクセスしてほしくない，またはそのバスサイクルを処理できない」という場合に要求される処理です(**図6**)．

ターゲットアボートはアクセスを拒絶する意味で通知するので，非常にクリティカルな場合に発行すべきであると考えます．ターゲットアボードを検出したイニシエータは，同じバスサイクルを再度発生させてはなりません．また，コンフィグレーションレジスタ空間のステータスレジスタ内のターゲットアボート受信フラグを1にセットします．

なお，**表2**を見ると，ターゲットアボートはDEVSEL# とTRDY# がともにディアサート状態で，STOP# のみアサート状態と示されていますが，これはDEVSEL応答がなく，いきなりSTOP# がアサートされるという意味ではありません．あくまでDEVSEL応答があったうえで，DEVSEL# のディアサートと同時にSTOP# をアサートされた場合です．

1.5 バスアービトレーションのしくみ

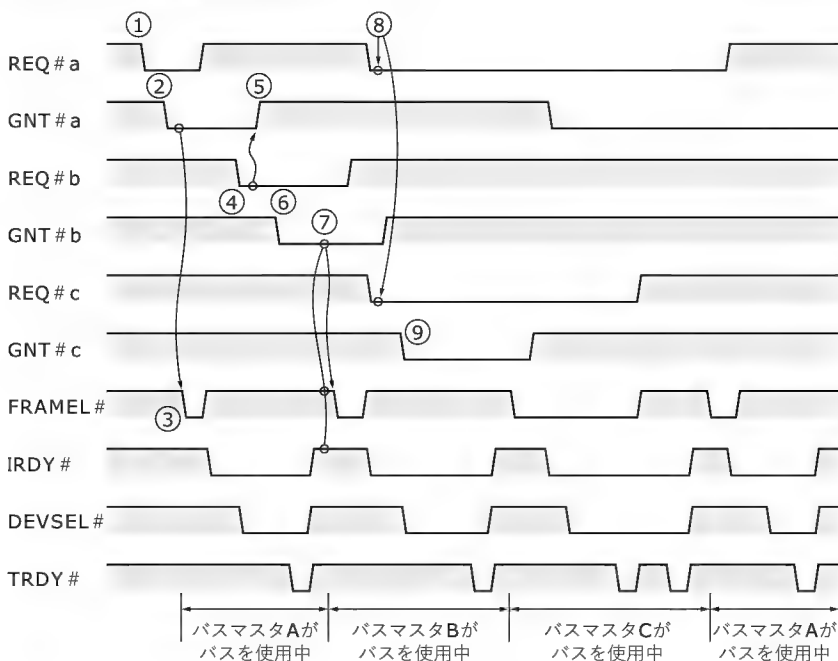
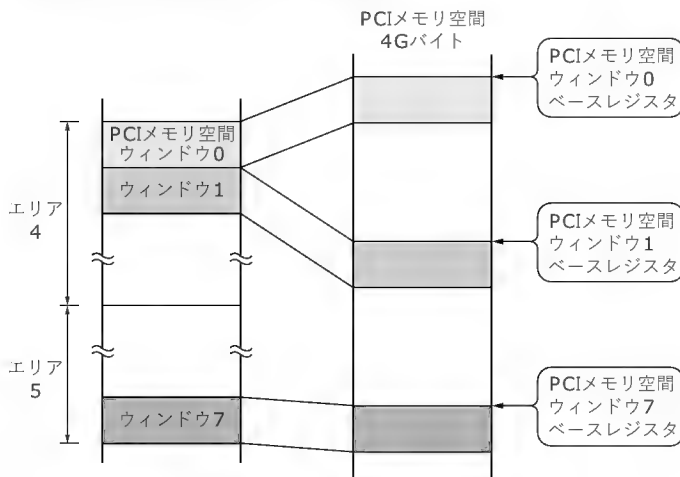
● バスアービタの動作

PCIバスでは，バスマスタデバイスからバスアービタに対して，バスの制御権を要求する信号としてREQ# が，逆にバスアービタからバスマスタデバイスに対して，バスの制御権を与える信号をGNT# として定義されています．

図7にバスの制御権とバスの動作を示します．電源投入直後，バス制御権がデバイスAからバスアービタへ要求されたとします(REQ#a アサート：**図7**①)．バスアービタは三つのバスマスタからのリクエストを確認し，Aからのみバス制御権の要求があったことを認識します．そこでバスアービタはデバイスAにバスの制御権を与えます(GNT#a アサート：**図7**②)．デバイスAはバスの制御権を取得したことを判定して，トランザクションを開始すると同時に，バスの制御権を取り下げます(REQ#a ディアサート：**図7**③)．デバイスAがバスを使っている間に，今度はデバイスBからバスの制御権が要求されました(REQ#b アサート：**図7**④)．バスアービタはデバイスAに対してバスを明け渡すように指示する(GNT#a ディアサート：**図7**⑤)とともに，次のクロックでデバイスBに対してバスの制御権を与えます(GNT#b アサート：**図7**⑥)．しかし，デバイスBはすぐにはバスを使えません．デバイスAがバスを使っているからです．バスアービタからバスの制御権を得ても，先にバスを使っているデバイスがバスを開放しないと，次のデバイスがバスを使うことはできません．バスが開放されたかどうかは，バスがアイドル状態(FRAME# とIRDY# が共に“H”レベル)になったかどうかで判定します．デバイスAがバスを開放してから，デバイス

〔**図5**〕ディスコネクトのタイミング

〔**図6**〕ターゲットアボートのタイミング

〔**図7**〕バスの制御権とバスの動作

〔図8〕PCIメモリ空間のマッピング仕様

Bはバスを使うことができます(**図7**⑦).

すると，今度はデバイスAとデバイスCが同時にバスの制御権を要求してきました(**図7**⑧). しかし，デバイスAは先ほどバスを使ったばかりです．そこでバスアービタは，デバイスCに対してバスの制御権を与えます(**図7**⑨).

● **バスパーキング**

バス制御権を同時に要求された場合でも，バスアービタは任意のアルゴリズムでバスの制御権を制御していきます．ここでもし，どのバスマスタデバイスからもバスの制御権が要求されなかった場合は，PCIバスはどのように動くのでしょうか．

PCIバスの規格では，ADバスやC/BE#，PAR信号にはプルアップ信号を入れないように指示されています．よって，誰もバスを使わないとなると，そのままではバスがフローティング状態になり好ましくありません．

そこで，実際にはバスを使う必要はないが，バスをフローティング状態にさせないために，任意の値でバスをドライブしておくという操作を行います．このような動作をバスパーキングと呼びます．問題は，誰(どのバスマスタデバイス)にバスをドライブさせるかです．

バスパーキングをさせるデバイスを固定的に決めておくという方法もあるでしょう．たとえば，PCIバスを搭載したシステムで必ず存在するはずのホストブリッジ(2段目以降のPCIバスの場合は上位PCIバスへのブリッジデバイス)にバスパーキングをさせるのです．もう一つの代表的な方法は，直前までバスを使用していたデバイスに，バスパーキングさせる方法です．アービタとしては，他のデバイスからバスの制御権要求がなければ，そのデバイスのGNT#をアサートしたままにすればよいからです．一般的にはこの方法を採用しているバスアービタが多いようです．

2 PCIホストコントローラの仕様

PCIバスの概要を説明したところで，本PCIホストコントローラの仕様について解説します．

〔表3〕PCIアドレス空間のアドレスマッピング

エリア	物理アドレス	論理アドレス(P2領域)	名　称	容　量(バイト)
0～3	0000_0000h ～	A000_0000h ～	メインメモリ	256M
4	1000_0000h ～	B000_0000h ～	PCIメモリ空間ウィンドウ0	16M
	1100_0000h ～	B100_0000h ～	PCIメモリ空間ウィンドウ1	16M
	1200_0000h ～	B200_0000h ～	PCIメモリ空間ウィンドウ2	16M
	1300_0000h ～	B300_0000h ～	PCIメモリ空間ウィンドウ3	16M
5	1400_0000h ～	B400_0000h ～	PCIメモリ空間ウィンドウ4	16M
	1500_0000h ～	B500_0000h ～	PCIメモリ空間ウィンドウ5	16M
	1600_0000h ～	B600_0000h ～	PCIメモリ空間ウィンドウ6	16M
	1700_0000h ～	B700_0000h ～	PCIメモリ空間ウィンドウ7	16M
6	1800_0000h ～	B800_0000h ～	SH-4ブート用フラッシュメモリ	16M
	1900_0000h ～	B900_0000h ～	(予約)	16M
	1A00_0000h ～	BA00_0000h ～	PCI I/O空間ウィンドウ	16M
	1B00_0000h ～ 1BFF_FFFFh	BB00_0000h ～ BBFF_FFFFh	PCIバス制御レジスタ空間ほか	16M

● **PCIバス空間のマッピング**

PCIバスのメモリ空間とI/O空間は4Gバイトのアドレス空間があります．I/O空間は実質的に64Kバイトの空間しか使われていません．しかしメモリ空間はSH-4の物理アドレスである448Mのサイズを超えているので，そのままではマッピングできません．そこで，PCIメモリ空間をマッピングするエリアとして，エリア4と5をさらに四つずつに分割し，それぞれベースアドレスを設定することで4Gバイト中の任意の16Mバイトをアクセスできるようにします．メモリウィンドウは0から7まで最大八つで，それぞれ連続したアドレスになるようベースアドレスを設定すれば，最大128Mバイトをリニアにアドレッシングできます(**図8**).

PCIバスのメモリ空間が4Gバイトあるとはいえ，実際にはそのすべてにデバイスが実装されることはありません．128Mバイトの空間があれば，よほど広いメモリ空間を使用するデバイスを使わないかぎり，一般的な使用でサイズが足りなくなることはないでしょう．

またPCIのI/O空間やその他の制御レジスタ空間は，エリア6を使うことにします．**表3**にPCIバス空間のアドレスマッピングを，**表4**にPCIバス制御レジスタを示します．

● **ホストコントローラに必須の機能**

ターゲットデバイスやバスマスタデバイスと異なり，ホストコ

〔表4〕PCIバス制御レジスタ一覧

オフセット	名　称
+000h	リビジョンID
+004h	PCIブリッジコントロール
+018h	コンフィグレーションアドレスレジスタ
+01Ch	コンフィグレーションデータレジスタ
+020h	割り込みレベルレジスタ
+024h	割り込みマスクレジスタ
+028h	割り込みステータスレジスタ
+040h	PCIメモリ空間ウィンドウ0ベースレジスタ
+044h	PCIメモリ空間ウィンドウ1ベースレジスタ
+048h	PCIメモリ空間ウィンドウ2ベースレジスタ
+04Ch	PCIメモリ空間ウィンドウ3ベースレジスタ
+050h	PCIメモリ空間ウィンドウ4ベースレジスタ
+054h	PCIメモリ空間ウィンドウ5ベースレジスタ
+058h	PCIメモリ空間ウィンドウ6ベースレジスタ
+05Ch	PCIメモリ空間ウィンドウ7ベースレジスタ
+07Ch	PCI I/O空間ベースレジスタ

ントローラには次の機能が必須となります．

- PCIバスクロック生成/リセット制御
- コンフィグレーションサイクル発生機能
- バスアービトレーション機能

PCIバスクロックやリセット信号の制御は，ホストが実装して各PCIデバイス/拡張スロットに供給します．今回のホストコントローラでは，PCIバスクロックは電源オンと同時に供給を開始し続けます．PCIリセットはホストCPUから制御可能なように，リセットコントロールレジスタを実装しました．

コンフィグレーションとは，PCIバスのプラグ&プレイシステムの要となるシステムで，各PCIデバイスがハードウェア資源の競合などを起こさないよう，それぞれリソースを割り当てる作業を呼びます．

具体的には，**表1**に示したコンフィグレーションサイクルを発生させることになりますが，コンフィグレーションサイクル時に，PCIデバイスを選択するのに，IDSELという信号を使います．

またバスアービトレーションは，FPGAの信号ピン数の関係から小容量のCPLDを別途実装し，そこにバスアービタを実装しました．

〔写真1〕市販のPCIバックプレーン（インタフェース社製）

● PCIバックプレーンとPICMG仕様

プロセッサボードはPCIのカードエッジコネクタを実装した，あくまで1枚のPCIボードです．このPCIバスにほかのPCIデバイスを接続するには，PCIバックプレーンが必要です．

ここでは市販されているPCIバックプレーンとして**写真1**に示すものを使いました．このバックプレーンは5スロットあり，スロット0がプロセッサボード専用で，残りのスロット1～4に合計4枚のPCIボードを実装することが可能です．

プロセッサスロットの信号ピンの配置は，通常のPCIボードと若干異なり，**図9**のようにクロックやバスアービトレーション信号がポイント-ポイントで配線されています．この信号ピンの配置はPICMG仕様で決まっているものです．クロックはまったく同じ信号ですが，各スロットへ位相を合わせたクロックを供給するために，クロックドライバから独立して出力するよう規定されています．

今回設計したプロセッサボードのPCIカードエッジは，通常のPCIボードとPICMG仕様の両方に対応可能です．**写真1**のバックプレーンでプロセッサスロットに差し込む場合には，PCIカードエッジをPICMG仕様にします．

プロセッサスロットからスロット1にはREQA#/GNTA#/CLKA，スロット2にはREQB#/GNTB#/CLKBが，以降スロット4まで配線されています．さらにスロット1のIDSELにはAD[31]が，スロット2にはAD[30]がダンピング抵抗を介して配線されています．これはコンフィグレーションサイクル時にADバスに出力される信号と関係してきます．

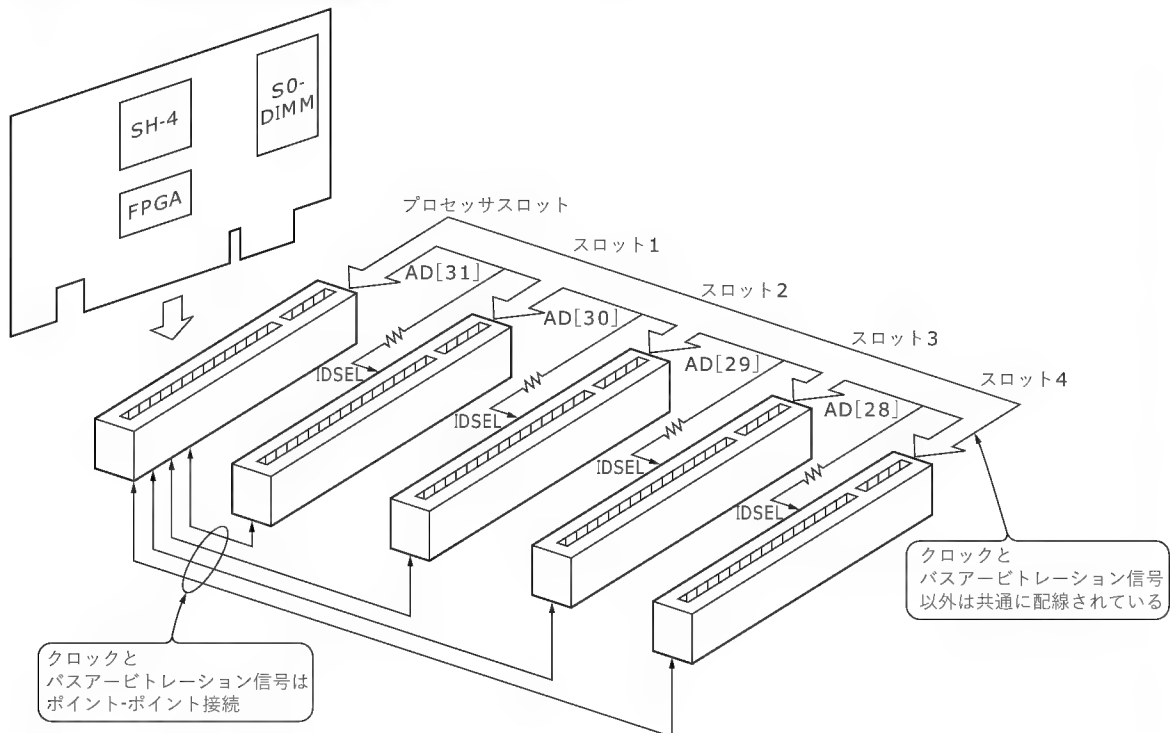
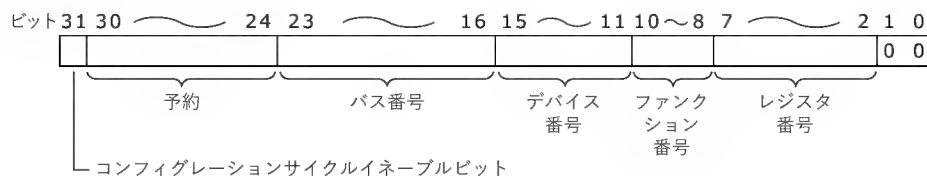
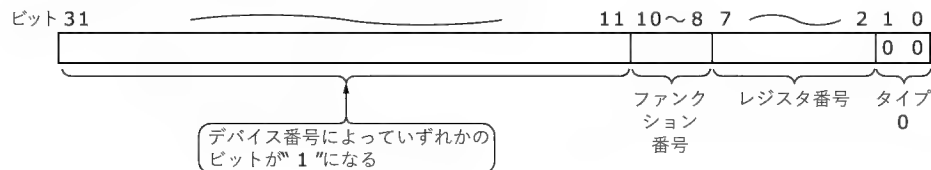
● コンフィグレーションとIDSEL

オリジナルアーキテクチャなので，既存の概念にとらわれることはないのですが，新しい概念は理解するのに苦労します．後から次々追加/拡張されたような仕様は見るに耐えませんが，よく整理され練られた仕様なら，それをあえて替える必要もありません．

そこで今回のホストコントローラのコンフィグレーションサイクルまわりの仕様も，PC/AT互換機のレジスタ仕様にならうことにしました．

表4にあるコンフィグレーションアドレスレジスタのフォーマットを**図10**に示します．このレジスタにアクセス先のPCIデバイスのバス番号，デバイス番号，ファンクション番号，レジスタ番号をセットし，コンフィグレーションデータレジスタを読み書きすることで，PCIバス上にコンフィグレーションサイクルが発生します．

なお，コンフィグレーションアドレスレジスタの最上位ビットはコンフィグレーションサイクルイネーブルビットになっていて，このビットが“0”のときは，コンフィグレーションデータレジスタをアクセスしてもコンフィグレーションサイクルは発生しません．

〔図9〕プロセッサスロットと各スロットの信号の配線

〔図10〕コンフィグレーションアドレスレジスタ

〔図11〕コンフィグレーションサイクル時のアドレスフェーズのアドレス値(タイプ0)

バス番号0でコンフィグレーションサイクルが発生した場合のADバスのアドレスフェーズに出力される信号を**図11**に示します．ビット10〜8はファンクション番号のデコードで，ビット7〜2はレジスタの選択で試用します．

そして，アクセスするPCIデバイスのデバイス番号が31の場合はビット31が，デバイス番号が11のときはビット11が"1"になります．つまり指定されたデバイス番号に該当するビットのみが"1"になります．

これを**図9**と照らし合わせてみてください．スロット1に差し込んだPCIデバイスを選択するためには，ビット31に"1"が立つように，デバイス番号を31と指定すればよいことがわかります．

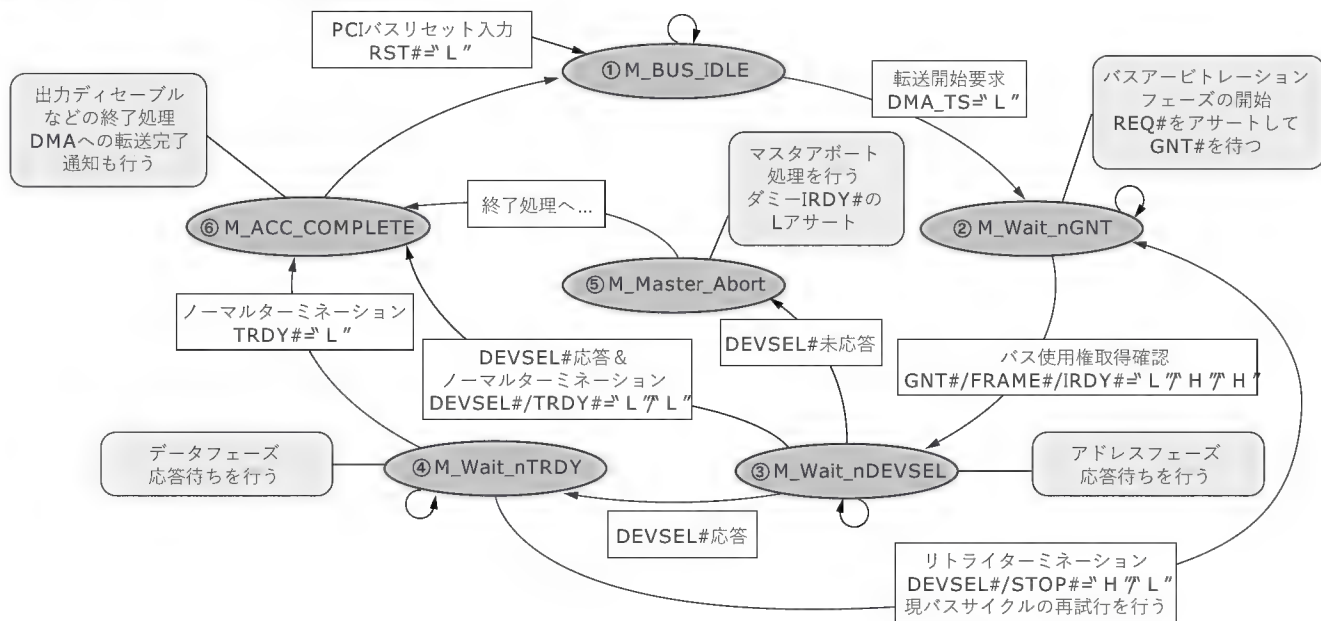
3 イニシエータのステートマシン

それでは，PCIホストコントローラとしておもに使われる，イニシエータ機能の制御回路について説明します．

● **シングル転送対応のイニシエータ**

今回設計するイニシエータは，データ転送がシングル転送のみという，もっとも基本的なイニシエータを設計します．

図12に今回設計したイニシエータシーケンサのステートマシンの状態遷移図を示します．合計六つのステートで，バスアービトレーションフェーズ，アドレスフェーズ，そしてデータフェーズを処理します．

〔図12〕イニシエータシーケンサの状態遷移図

〔リスト1〕M_BUS_IDLEステート

```
when M_BUS_IDLE =>                              -- イニシエータシーケンス・アイドルステート

    if  ( MASTER_EN = '1' and nSYNC_TS = '0' ) then
                                                -- バスマスタ有効でDMACのバスサイクル開始
                                                   要求を確認したら
         MAS_ACTIVE     <= pASSERT ;            -- イニシエータステート起動フラグのセット
                                                -- このフラグは一連のシーケンスが完了するまで
                                                -- セットしつづけられる

         -- バスリクエスト要求開始
         nREQ_Port      <= nASSERT ;            -- REQ#をアサート
         M_NEXT_STATE   := M_Wait_nGNT ;        -- GNT#アサート待ちステートに移行

    else
         M_NEXT_STATE   := M_BUS_IDLE
    end if ;
```

実際のバストランザクションは，バスマスタデバイスに内蔵されているダイレクトメモリアクセスコントローラ(以下DMAC)によるDMACの転送開始要求に対して，このバスマスタステートマシンが応答することによって開始されます．

イニシエータシーケンサは転送開始要求によってバストランザクションを開始し，ターゲットデバイスからのターミネーション応答で完了します．トランザクションが完了すると，DMACへ1ワードのデータ転送が終わったことを通知し，次の転送開始要求に備えます．

各ステートごとに処理内容を解説していきます．

● **M_BUS_IDLEステート**

このステートでは，DMACからの転送開始要求を待ち続けます(**リスト1**)．また，PCIバス側からのRST#信号アサートによるバスリセットが行われると，無条件にこのステートに初期化されます．

イニシエータのステートマシンの動作開始条件には，コンフィグレーションレジスタ内に用意されているバスマスタイネーブルビットがセットされていることも重要です．このビットがセットされていない間は，イニシエータシーケンサを起動してはなりません．

転送開始要求は，DMACコントローラからのDMA_TS信号のアサートか，またはこの信号をトリガにして一連のデータ転送が完了するまで開始要求を保持しつづけるストローブの同期信号(SYNC_TS)がHレベルであることで認識します．

バスマスタイネーブルビットがセットされ，DMACからの転送開始要求を認識すると，このステートではバスの制御権を要求するためにREQ#信号をアサートします．そしてバスアービタからのGNT#信号アサートを待つために，M_Wait_nGNTステートへ移行します．

また，シーケンサ内にあるMAS_ACTIVE信号は，REQ#のアサートと同時にセットすることで，イニシエータシーケンサがM_BUS_IDLEステート以外の動作状態にあることを示す信号として使います．

この信号はたとえば，ターゲットシーケンサ内のDEVSEL#応答の条件に入力して，同一デバイス内部でのバスサイクルを無視するような使い方も可能です．つまりバスマスタとしてア

〔リスト 2〕M_Wait_nGNT ステート

```
when M Wait nGNT =>

-- Bus grant request to PCI Bus arbitor.
    if  ( nGNT = '0'and nFRAME I /= '0' and nIRDY I /= '0' ) then -- and nSTOP /= '0') then
                                                       -- GNT# がアサートされていること
                                                       -- FRAME#/IRDY# がディアサート状態（バスアイドル）で
                                                       -- あれば以下の論理を実行する
    -- バスリクエスト要求終了
         nREQ_Port <= nNEGATE ;                        -- REQ ディアサート

    -- アドレスフェーズ開始
         nFRAME O  <= nASSERT ;                        -- FRAME# アサート
         FRAME OEN <= pASSERT ;                        -- FRAME# ポートのドライブ開始

    -- C/BE*[3:0] encode.
         if  ( DMA RD = '1' ) then                     -- **** リードサイクルであれば **** --
              nC BE Port     <= PCI MemReadCycle ;     -- 0110 : メモリリードコマンドをセットする
         else                                          -- **** ライトサイクルであれば **** --
              nC BE Port     <= PCI MemWriteCycle ;    -- 0111 : メモリライトコマンドをセットする
         end if ;
         CBE OEN             <= pASSERT ;              -- C/BE# ポートのドライブ開始

    -- PCIAD access address output port encode.
         PCIAD O Port (31 downto  2) <= DMA AD (31 downto 2) ;     -- AD バスにアクセス先アドレスをセット
         PCIAD O Port ( 1 downto  0) <= “00” ;         -- リニアアドレッシングモードのセット
         PCIAD OEN      <= pASSERT ;                   -- AD ポートのドライブ開始
         dPAR O OEN     <= pASSERT ;                   -- PAR ポートのドライブ開始（実際には 1 クロック遅れてドライブ）
         M NEXT STATE   := M Wait nDEVSEL ;            -- DEVSEL# 応答待ちステートへ移行する

    else

         IRDY OEN       <= pNEGATE ;                   -- IRDY# ポートのドライブ終了（リトライで戻ってきたときのため）
         M NEXT STATE   := M Wait nGNT ;               -- バス制御件取得 & バスアイドルまで待つ
    end if;
```

クセスしようとしたアドレスが自分に割り当てられたアドレスだった場合には，DEVSEL# 応答しないようにするわけです．

- **M_Wait_nGNT ステート**

このステートは，バスの制御権を取得するまで待つステートです（**リスト 2**）．先の M_BUS_IDLE ステートでアサートされたバス制御権要求信号である REQ# に対して，一般的にホストブリッジ内に実装されているバスアービタは，システムにインプリメントされているアービトレーションの手順に基づいて，バス制御権である GNT# 信号をアサートします．各バスマスタは，GNT# がアサートされたことを確認して，自分にバス制御権が与えられたことを判定できます．

しかしバスの制御権が取得できたとしても，すぐにトランザクションを開始できるわけではありません．アドレスフェーズを開始するには，バスがアイドル状況であることが必要です．バスアイドル状態とは，FRAME# が“H”，IRDY# が“H”という状態です．このステートではバス制御権の取得（GNT# =“L”）とバスアイドル状態（FRAEM#=“H”，IRDY#=“H”）を同時に if 文で判定しています．

トランザクションの開始は，次のような処理が必要です．

- REQ# をディアサート
- FRAME# をアサートして信号ドライブ開始
- C/BE# にバスコマンドを出力し信号ドライブ開始
- AD バスにアドレスを出力し信号ドライブ開始

PCI バスのアドレス空間は 32 ビット長ですが，データバス幅が 32 ビットなので，アクセス先アドレスは 1 ワード/4 バイト単位となり，AD[31：02]の 30 ビット分にアクセス先アドレスをセットします．下位の AD[01：00]の 2 ビットには 00 をセットします．下位 2 ビットのこの値は，バースト転送時にリニアアドレッシングモードとして用いられます．今回設計するイニシエータはメモリリード/ライトトランザクションかつシングル転送専用なのですが，一般的にメモリサイクル時の下位 2 ビットは 00 にします．

これらの処理を行った後，ターゲットデバイスからの DEVSEL# 応答を待つ M_WAIT_nDEVSEL ステートに移行します．

- **M_Wait_nDEVSEL ステート**

このステートは，ターゲットデバイスからの DEVSEL# 応答を待つステートです（**リスト 3**）．

このステートに移行して真っ先に行わなければならない処理は，アドレスフェーズからデータフェースへの切り替えです．処理内容を次に示します．

- FRAME# をディアサートする（シングル転送）
- AD バスは，リードサイクルであればイニシエータが入力デバイスとなるため AD バスを入力に切り替える．ライトサイクルあれば出力デバイスとして継続して AD バスをドライブし，書き込みデータを出力する
- C/BE# にバイトイネーブル信号を出力する
- IRDY# をアサートしてドライブ開始

これらの処理を行った後，ターゲットデバイスからの DEVSEL# 応答待ち状態になります．

〔リスト3〕M_Wait_nDEVSELステート

```
-- ********** Wait nDEVSEL will be asserted ********** --

when M_Wait_nDEVSEL =>

    nREQ_Port <= nNEGATE ;                          -- REQ#ディアサート
    nFRAME_O  <= nASSERT ;                          -- FRAME#ディアサート
    IRDY_OEN  <= pASSERT ;                          -- IRDY#ドライブ開始

    if ( DMA_RD = '1' ) then                        -- リードサイクルの場合の処理
         PCIAD_OEN     <= pNEGATE ;                 -- ADバスのドライブ終了
         dPAR_O_OEN    <= pNEGATE ;                 -- PARポートのドライブ終了（実際には1クロック遅れてドライブ）
         nC_BE_Port    <= "0000" ;                  -- バイトイネーブルは全てアサート．32ビット長
    else                                            -- ライトバスサイクルの場合の処理
         PCIAD_O_Port<= DMA_DI ;                    -- ADポートのDMACからの書き込みデータをセットする
         nC_BE_Port    <= nDMA_BE ;                 -- ライトのときはDMA_BE*でバイト単位のイネーブルを行う
    end if ;

    -- DEVSEL#応答のタイムアウトチェック
    if ( BusErr_Cnt = TimeOut ) then                -- もしDEVSEL#応答がなかった場合には・・・  ←Ⓐ
         BusErr_Cnt := ( others => '0' ) ;          -- DEVSEL#未応答チェックカウンタのクリア
         Receive_MA    <= '1' ;                     -- マスタアボート受信フラグのセット開始
         AERR          <= '1' ;                     -- アドレスフェーズ中のエラーがあったことを通知する
         nDMA_TA_Port  <= '0' ;                     -- あわせて，DMACへデータ完了通知を行う
         M_NEXT_STATE := M_Master_Abort ;           -- マスタアボート処理ステートへ移行

    elsif ( nDEVSEL = '0' ) then                    -- DEVSEL#応答を認識したら……
         BusErr_Cnt := ( others => '0' ) ;          -- DEVSEL#未応答チェックカウンタのクリア
         if ( nTRDY = '0' ) then                    -- 同時にTRDY#:ターゲットレディを認識した場合

              -- **** ノーマルターミネーション時 **** --
              if ( DMA_RD = '1' ) then
                   DMA_DO    <= PCIAD_I ;           -- もしリードサイクルならAD値を取り込む
              end if ;

              nDMA_TA_Port <= '0' ;                 -- DMACに転送終了通知を発行
              PCIAD_OEN      <= nNEGATE ;           -- ADポートのドライブ終了
              CBE_OEN        <= nNEGATE ;           -- C/BE#ポートのドライブ終了
              dPAR_O_OEN     <= pNEGATE ;           -- PARポートのドライブ終了（実際には1クロック遅れてドライブ）
              FRAME_OEN      <= nNEGATE ;           -- FRAME#ポートのドライブ終了
              nIRDY_O        <= nNEGATE ;           -- IRDY#のディアサート
              M_NEXT_STATE := M_ACC_COMPLETE ;      -- アクセス終了ステートへ移行

         else
              -- **** リトライターミネーション時 **** --
              if ( nSTOP = '0' ) then               -- **** リトライを受信した場合の処理 **** --
                   nREQ_Port      <= nASSERT ;      -- REQ#アサート．バスリクエスト要求開始
                   PCIAD_OEN      <= nNEGATE ;      -- ADポートのドライブ終了
                   CBE_OEN        <= nNEGATE ;      -- C/BE#ポートのドライブ終了
                   dPAR_O_OEN     <= pNEGATE ;      -- PARポートのドライブ終了（実際には1クロック遅れてドライブ）
                   FRAME_OEN      <= nNEGATE ;      -- FRAME#ポートのドライブ終了
                   nIRDY_O        <= nNEGATE ;      -- IRDY#のディアサート
                   M_NEXT_STATE := M_Wait_nGNT ;    -- GNT#アサート待ちステートへ移行

              else
              -- **** ターゲットレディ待ち **** --
                   M_NEXT_STATE := M_Wait_nTRDY ;   -- TRDY#が来ない場合は
                                                    -- TRDY#アサート待ちステートへ移行
              end if ;
         end if ;

    else
         -- このステートに来た最初の1クロックは必ずここを実行する
         nIRDY_O        <= nASSERT ;                -- IRDY#アサート
         BusErr_Cnt := BusErr_Cnt + 1;              -- DEVSEL#応答が無ければタイムアウトカウントを継続する
         M_NEXT_STATE := M_Wait_nDEVSEL;            -- DEVSEL#応答があるまで待つ
    end if ;
```

▶マスタアボート

この次からが，さまざまなターゲットデバイスを想定した処理が必要になります．たとえばアクセスしたアドレスが，必ずしもデバイスの存在するアドレスとはかぎりません．つまり，必ずDEVSEL#応答が返ってくるとはかぎらないわけです．場合によってはデバイス未実装領域へのアクセスした場合も考えられるため，DEVSEL#応答待ちのタイマを起動します．

リスト3のⒶがその部分であり，ここでタイムアウトチェックを行います．PCIバスクロックで4クロックを経過したにもかかわらずDEVSEL#応答がなければ，アクセス先のアドレスはデバイス未応実装領域であるとして，コンフィグレーションレジスタのステータスレジスタ内に，マスタアボート通知フラグ用をセットする信号であるReceive_MA信号をセットします．そしてM_Master_Abortステートへ移行します．

高速応答デバイスのターミネーション

また，非常に高速なターゲットの場合は，DEVSEL#応答と

〔リスト 4〕 **M_Wait_nTRDY ステート**

```
when M Wait nTRDY =>

    if  ( nTRDY = '0' ) then                              -- TRDY# のアサートを確認したら以下の処理を行う.
         if  ( DMA RD = '1' ) then
              DMA DO     <= PCIAD I ;                     -- もしリードサイクルなら AD 値を取り込む
         end if ;
         nDMA TA Port   <= '0';                           -- DMAC に転送終了通知を発行
         PCIAD OEN      <= nNEGATE ;                      -- AD ポートのドライブ終了
         CBE OEN        <= nNEGATE ;                      -- C/BE# ポートのドライブ終了
         dPAR O OEN     <= pNEGATE ;                      -- PAR ポートのドライブ終了（実際には 1 クロック遅れてドライブ）
         FRAME OEN      <= nNEGATE ;                      -- FRAME# ポートのドライブ終了
         nIRDY O        <= nNEGATE ;                      -- IRDY# のディアサート
         M NEXT STATE := M ACC COMPLETE ;                 -- アクセス終了ステートへ移行

    elsif  ( nSTOP = '0' ) then                           -- **** なんらかのアボート処理を受信!! **** --

         -- ターゲットアボート＆リトライ共通のターミネーション処理
         PCIAD OEN      <= nNEGATE ;                      -- AD ポートのドライブ終了
         CBE OEN        <= nNEGATE ;                      -- C/BE# ポートのドライブ終了
         dPAR O OEN     <= pNEGATE ;                      -- PAR ポートのドライブ終了（実際には 1 クロック遅れてドライブ）
         FRAME OEN      <= nNEGATE ;                      -- FRAME# ポートのドライブ終了
         nIRDY O        <= nNEGATE ;                      -- IRDY# のディアサート

         if  ( nDEVSEL ='1' ) then                        -- **** ターゲットアボートを受信した時 **** --
              Receive TA      <= '0';                     -- ターゲットアボート受信フラグのセット開始
              DERR            <= '1';                     -- データフェーズ中のエラーがあったことを通知する
              nDMA TA Port   <= '0';                      -- あわせて，DMAC へデータ完了通知を行う
              M NEXT STATE := M ACC COMPLETE ;            -- アクセス終了ステートへ移行

         else                                             -- **** リトライを受信した場合の処理 ****
              nREQ_Port       <= nASSERT ;                -- REQ# アサート．バスの再試行開始
              M NEXT STATE := M Wait nGNT ;               -- GNT# アサート待ちステートへ移行
                                                          -- 注)
                                                          --      リトライシーケンスなので，DMAC 側へ
                                                          --      DMA TA* ：転送終了通知は出さない．
         end if ;

    else
         M NEXT STATE := M Wait nTRDY
    end if ;
```

同時に TRDY# をアサートしてくるデバイスも考えられます．また，STOP# をアサートしてトランザクションを打ち切ってくるかもしれません．これらターゲット側の要因でトランザクションを終了するターゲットイニシエーテッドターミネーション処理も同時に確認します．

具体的には，DEVSEL# 応答を認識すると同時に TRDY# と STOP# の両信号を確認し，以下のような処理を行います．

TRDY# がアサートされていれば，DEVSEL# 応答と同時にデータ転送も完了したとして，DMA_TA を L にアサートして DMAC に転送完了通知を出します．また，リードサイクルであれば，AD バスの値を DMAC 側データ出力バスに取り込みます．そしてアクセス完了処理ステートである M_ACC_COMPLETE ステートに移行します．なおこのとき，STOP# がアサートされていてもそれを無視します．今回のイニシエータはシングル転送のみサポートするので，TRDY# と STOP# の両方がアサートされてもディスコネクトとして認識しません．

▶リトライ処理

TRDY# が“H”で STOP# が“L”であればリトライ要求であると判断して，リトライターミネーション処理を行います．まず，即座に次のバス制御権を要求するために REQ# をアサートします．また，アサートしていた IRDY# をディアサートします．AD バス，C/BE# のバスドライブも終了します．FRAME# はすでにディアサート状態なので，ここで信号のドライブを終了しても問題ありません．しかし IRDY# はアサート状態です．この信号もサスティンドトライステート処理が必要なので，このステートでハイインピーダンスにするわけにはいきません．ディアサート状態にして，バスアービタによる GNT# アサートを待つための M_Wait_nGNT ステートに移行します．リトライ時の IRDY# のドライブ終了処理は，M_Wait_nGNT 内で行っています．

▶ごく一般的なターゲットデバイスの場合

TRDY# が“H”，STOP# が“H”であれば，DEVSEL# 応答があってから数クロック後に TRDY# がアサートされる，ごく一般的な動作をするターゲットデバイスであるとわかります．ターゲットのデータ転送の準備が整う，つまり TRDY# がアサートされるまで待つ M_Wait_nTRDY ステートに移行します．

● **M_Wait_nTRDY ステート**

ターゲットのデータ転送の準備が整うまで待つステートです（**リスト 4**）．

ターゲットイニシエーテッドターミネーションには 4 種類があります．すなわち通常の転送完了，リトライ，ディスコネクト，そしてターゲットアボートです．今回設計するイニシエータはシングル転送しか対応していないので，このうちのディスコネクト動作については考慮する必要はありません．また，通常の転送完了では STOP# はアサートされません．

▶転送完了

TRDY# がアサートされれば通常のターミネーション処理を行います．リードサイクルであれば AD バスのデータを取り込み，DMAC に転送完了通知を出します．また IRDY# をディアサートし，FRAME# や AD バス，C/BE# のドライブを終了し，アクセス完了処理ステートである M_ACC_COMPLETE ステートに移行します．また，ここでも TRDY# と同時に STOP# がアサートされていた場合，それを無視してかまいません．

▶ターゲットアボート

TRDY# がディアサート状態で，STOP# がアサートされた場合，リトライもしくはターゲットアボート処理が必要です．まず共通のターミネーション処理として，FRAME# や AD バス，C/BE# のドライブを終了し，IRDY# をディアサートします．

さらに DEVSEL# がディアサートされていた場合は，ターゲットアボードと判定します．イニシエータがターゲットアボートを検出すると，次のような処理を行います．

- ターミナルアボート受信フラグセット用信号（Receive_TA）をセットする
- IRDY# をディアサートする
- FRAME# や AD バス，C/BE# のドライブを終了する
- DMA_TA をアサートして DMAC へデータ転送が終了したことを通知する
- DMAC へデータフェーズ中にエラーが発生したことを示す DERR フラグセット用信号をアサートする

これらの処理を行った後に，トランザクション完了ステートである M_ACC_COMPLETE ステートへ移行します．

このターゲットアボートは，ターゲット側は必ず直前に DEVSEL# 応答をした後に発生します．よって M_Wait_nDEVSEL ステートでは判定できません．DEVSEL# 応答なしで，いきなり STOP# だけがアサートされることはありません．

▶リトライ

DEVSEL# と STOP# がアサート状態で，IRDY# がディアサート状態の場合は，リトライと判定します．

リトライの場合は再度同じバスコマンドとアドレスでトランザクションを再実行しなければなりません．よってリトライを検出したときは，即座にバスの制御権を要求して再度同じトランザクションを処理します．しかしターゲットアボートは同じトランザクションを再度実行してはなりません．ターゲットアボートが発生したことを示しエラーフラグを立てて，すみやかにターミネーション処理に入ります．

● M_Master_Abort ステート

M_Wait_nDEVSEL ステートで，4 クロックが経過しても DEVSEL# 応答がなかった場合に遷移してくるステートです（**リスト 5**）．

イニシエータの制御する FRAME# や IRDY# には，その制御に基本的なルールがあります．両方の信号を同時にアサートしたりディアサートしてはいけません．また，FRAME# を一度アサートしたら，IRDY# をアサートせずに FRAME# をディアサートしてはいけません．

ここで，たとえば FRAME# が“L”，IRDY# が“H”の状態でマスタアボートを検出しても，IRDY# をアサートしないまま FRAME# だけをディアサートしてトランザクションを終了することはできません．この場合はダミーでも一度 IRDY# をアサートして，次のクロックでディアサートします．

今回設計したステートマシンは，M_WAIT_nDEVSEL ステートに入って最初に IRDY# をアサートしているので，ダミーとして IRDY# をアサートする必要はありません．マスタアボードを検出したら，このステートで IRDY# をディアサートし，FRAME# や AD バス，C/BE# のドライブを終了します．さらに IRDY# のドライブ終了は，次のステートである M_ACC

〔リスト 5〕M_Master_Abort ステート

```
when M_Master_Abort =>

    PCIAD_OEN       <= nNEGATE ;                    -- AD ポートのドライブ終了
    CBE_OEN         <= nNEGATE ;                    -- C/BE# ポートのドライブ終了
    dPAR_O_OEN      <= pNEGATE ;                    -- PAR ポートのドライブ終了（実際には 1 クロック遅れてドライブ）
    FRAME_OEN       <= nNEGATE ;                    -- FRAME# ポートのドライブ終了
    nIRDY_O         <= nNEGATE ;                    -- IRDY# のディアサート
                                                    -- 最低 1 クロック以上の IRDY# アサートが存在したことになる．
    M_NEXT_STATE := M_ACC_COMPLETE ;                -- アクセス終了ステートへ移行
```

〔リスト 6〕M_ACC_COMPLETE ステート

```
when M_ACC_COMPLETE =>
    if(nSYNC_TS='1')then
        PCIAD_O_Port   <= ( others => '0' ) ;       -- AD ポートの F/F クリア
        IRDY_OEN       <= nNEGATE ;                 -- IRDY# ポートのドライブ終了
        nDMA_TA_Port   <= '1' ;                     -- DMAC へのデータ転送完了通知アクノレッジフラグのクリア
        MAS_ACTIVE     <= '0' ;                     -- イニシエータステート起動フラグのクリア
        AERR           <= pNEGATE ;                 -- アドレスフェーズエラー発生通知フラグのクリア
        DERR           <= pNEGATE ;                 -- データフェーズエラー発生通知フラグのクリア
        Receive_MA     <= pNEGATE ;                 -- マスタアボート受信フラグセット信号のクリア
        Receive_TA     <= pNEGATE ;                 -- ターゲットアボート受信フラグセット信号のクリア
        M_NEXT_STATE := M_BUS_IDLE ;
    end if;
```

_COMPLETE ステートで行います.

● **M_ACC_COMPLETE ステート**

すべてのイニシエータトランザクションの最後の処理を行うステートです(**リスト5**). このステートでは, 次のような処理を行います.

- 出力側 AD ポートのフリップフロップをゼロにクリア
- IRDY# の信号ドライブ終了

また, DMAC 側やエラーが発生した場合のステータスフラグセット用トリガの信号系も次のように処理します.

- DMAC 側への転送完了通知フラグ(nDMA_TA_Port)のリセット
- マスタ機能アクティブフラグ(MAS_ACTIVE)のリセット
- AERR(アドレスフェーズ中にエラーが発生)や DERR(データフェーズ中にエラーが発生)の二つの DMAC へのエラー状況通知フラグのリセット
- コンフィグレーションレジスタ内のマスタアボート受信フラグのセット信号(Receive_MA)をゼロに戻す
- コンフィグレーションレジスタ内のターゲットアボート受信フラグのセット信号(Receive_TA)をゼロに戻す

このように, ここまでに発行していた出力イネーブル信号やフラグセット系の信号を, DMAC からのデータ転送要求待ち状態にまで戻す操作を行っています.

● **パリティ信号の制御**

パリティ信号に関係する PAR 信号(dPAR_O_OEN)の出力イネーブル信号も本ステート内でクリアしております.

この信号は FRAME # や IRDY# などの信号よりも 1 クロック遅れてディアサートされるように, この信号を 1 段のシフトレジスタに入れたものを PAR 信号の出力イネーブルとして利用しています.

これにより, ターゲットデバイスへのライトサイクルにおけるデータフェーズ中の PAR は FRAME#/IRDY# ディアサート後 1 クロック遅れるところまでバス上に存在することになります. すでにご承知のとおり, PAR 信号は AD や C/BE# 信号と同じくトライステート信号ですから TRDY# 認識の 2 クロック後にハイインピーダンスに移行しても動作上, ならびに規格上は問題ありません.

これらの処理を行った後はリセットアサート直後の M_BUS_IDLE ステートに復帰し, 次のダイレクトメモリアクセスコントローラからのデータ転送要求待ち状態に戻ります.

4 ターゲットステートマシンとバスアービタ

● **ホストコントローラのターゲット機能**

ホストコントローラはイニシエータ機能だけで, ターゲットの機能がいらないというわけではありません. バスマスタデバイスがホストのメインメモリに対してバスマスタ転送するという場合は, ホストコントローラがターゲットの立場となります.

今回のシステムの場合, メインメモリは第 2 章で解説した SDRAM が該当します. SDRAM は SH-4 がメインメモリとしてもアクセスしているので, ターゲットとして応答したときに SH-4 が SDRAM をアクセスしていた場合は, SH-4 のアクセスが終了するまで待たなければなりません.

なお, ターゲットステートマシンについては誌面の都合で省略します.

● **バスアービタ**

すでに説明したように, 今回は別チップとして外付けにバスアービタを CPLD で実装しています.

バスアービタには, 複数のバスマスタデバイスからのバスの使用権要求を受け付け, その時々に応じて要求がきたバスマスタデバイスへバスの制御権を許可するものです. 複数のバスマスタから同時にバスの制御権要求がくることもあります. そのときにどのような手順で許可を与えるのかという方法については, PCI バスのスペックでは明確に規定されていません. しかし, 特定のバスマスタのみがバスを使い続けることがないよう, 順次リクエストに答えるように循環してバス制御権を与えることが推奨されています.

このバスアービトレーション方法をラウンドロビン方式といい, ラウンドロビン方式のバスアービタをラウンドロビン型バスアービタと呼びます. 今回のシステムでも, この方式を採用しました.

5 PCI バス割り込みコントローラ

● **INTA# ～ INTD# の PCI デバイス割り込み**

PCI バスには INTA# ～ INTD# の 4 本の割り込み信号線があります. PCI バスの割り込みは, レベル出力のため, 割り込み線を複数のデバイスが共有することが可能です. つまり, 割り込みプログラムは共有可能なように作成する必要があります.

PCI バスの INTA# ～ INTD# をどのようにホスト CPU に通知するかという点ですが, 今回はこの 4 本の割り込みを 1 本にまとめて出力し, INTA# ～ INTD# のどのラインが割り込みを出力しているかをステータスレジスタで取得できるようにしました. この割り込みを PCI デバイス割り込みと呼ぶことにします.

● **アボートエラー**

イニシエータステートマシンのところで説明しましたが, マスタがアクセスしたアドレスにデバイスが存在しなかった場合は, マスタアボートで終了します. また, そのアドレスにデバイスは存在したが, 何らかの理由でアクセスを拒否された場合はターゲットアボートで終了します.

リトライやディスコネクトが発生した場合は, イニシエータシーケンサは再度トランザクションを開始するよう設計しましたが, マスタアボートやターゲットアボートが発生した場合はエラーとみなし, イニシエータシーケンサは, その時点でバスアクセスを終了します.

〔リスト7〕作成したPCI BIOSのヘッダファイル

```
/*************************************************************
    PCI BIOS
*************************************************************/
#define PCI BIOSVer    0x0200    /* PCI BIOS バージョン Ver 2.00 */

#define PCI IoTopAdr   0xFD00         /* PCI I/O空間 64K-768 */
#define PCI MemTopAdr  0xB8000000     /* PCI 通常メモリ空間 */
#define PCI PreTopAdr  0xB4000000     /* PCI プリフェッチメモリ空間 */

#define PCI ISABridg   1    /* ISAバスブリッジの存在を考慮する場合は 1 */
#define PCI PERREn     1    /* パリティエラーをチェックする場合は 1 */
#define PCI SERREn     1    /* システムエラーをチェックする場合は 1 */
#define PCI LatencyTimer    0x40
                                 /* レイテンシタイマ 40h デフォルト */
#define PCI CacheLineSize   8
                            /* キャッシュラインサイズ 8ワード デフォルト */

/* PCI BIOS 諸情報取得データ構造 */
struct PCIBIOS INFO {
    int BIOSVer;                   /* PCI BIOSバージョン */
    unsigned char MaxBusNo;        /* 最大バス番号 */
};

/* PCIデバイス割り込み(INTAライン~INTDライン)マスクビット位置生成マクロ */
#define PCI IntLineBit(PCI IntLine)   1<<(PCI IntLine-1)

/* PCI BIOSバージョン&最大バス番号取得
引き数
    *Info     PCI BIOS 諸情報取得データ構造格納バッファアドレス
*/
void PCIBIOS GetInfo(struct PCIBIOS INFO *Info);

/* PCI BIOS コンフィグレーションレジスタリード
引き数
    PCI BusNo      バス番号
    PCI DevNo      デバイス番号
    PCI FuncNo     ファンクション番号
    PCI RegAdr     コンフィグレーションレジスタ番号
戻り値
    レジスタ読み出しデータ
*/
#define PCIBIOS CfgByteRead PCIBridge CfgByteRead
#define PCIBIOS CfgWordRead PCIBridge CfgWordRead
#define PCIBIOS CfgLongRead PCIBridge CfgLongRead

                    ~以下省略~
```

これらアボート状態はアクセスエラーとして，ホストCPUに割り込みで通知できると，より信頼性の高いシステムを構築できます．

● パリティエラー/システムエラー

さらにPCIバスには，データ転送時のパリティエラーを示すPERR#があります．またアドレスフェーズ時のアドレスパリティエラーをシステムエラーと呼び，SERR#という信号もあります．

これらのエラー信号を検出した場合にも，ホストCPUに対して割り込みを出力できる構造にしました．

今回はこれらアボート系のエラーとパリティ系のエラーを1本にまとめて，PCIコントロール割り込みと呼ぶことにします．

● 2レベルの割り込み出力

PCIデバイス割り込みとPCIコントロール割り込みをそれぞれホストCPUに出力するわけですが，第1章で説明したように，SH-4の割り込み入力は15レベルの優先順位付き割り込みが使えます．そこで優先順位に差をつけてホストに出力することにします．

それぞれの割り込み要因に対して独立してレベルを設定できるような設計でもよいのですが，PCIデバイス割り込みよりPCIコントロール割り込みのほうが緊急度が高いと判断し，次のような構造にしました．SH-4のIRLは4本ありますが，IRLレベル設定レジスタはビット3～1までの3ビットを実装し，PCIデバイス割り込みが発生した場合はビット0を"1"にして，PCIコントロール割り込みが発生した場合はビット0を"0"にして出力することで，PCIコントロール割り込みが設定されたレベルより一つだけ高いレベルで出力されるようにしました(IRL信号は負論理なのでレベル15はオール"L"になる)．

6 PCI BIOS

● PC/AT互換機のPCI BIOSに準拠

最後に，ここで設計したPCIホストコントローラ用に作成したPCI BIOSについて説明します．**リスト7**にPCI BIOSのヘッダファイルを示します．コンフィグレーションサイクルを制御するレジスタがPC/AT互換機のレジスタに準拠したものであるため，PCI BIOSのエントリもそれに準拠したスタイルになっています．

PCI BIOSに必要な機能は大きく分けて三つあります．もっとも頻繁に使うであろう，PCIバス番号/デバイス番号/ファンクション番号/レジスタアドレスを使ったPCIコンフィグレーションレジスタ空間へのアクセス機能，そしてベンダIDやデバイスID，もしくはクラスコードによるPCIデバイス検索機能，そしてもっとも重要かつ難しいのは，各PCIデバイスに対してコンフリクトなくリソースを割り当てていく初期化機能です．

今回作成したPCI BIOSはPCI-PCIブリッジにも対応し，ブリッジの下も次々検索してPCIバスツリー全体を初期化します．また，ISAバスブリッジにも対応し，レガシーI/Oが存在する可能性のあるI/Oアドレスは使用しないという設定も可能です．なお，PC/AT互換機用のVGAビデオカードをVGAで使用するつもりはないので，VGAパレットスヌープなどの機能は初期化していません．

バスマスタデバイスに対しては，すべてレイテンシタイマを40h，キャッシュラインサイズを8ワードに固定で割り当てています．

参考文献

1) TECH I Vol.3『PCIデバイス設計入門』，CQ出版(株)
2) OPENDESIGN No.7『PCIバスの詳細と応用へのステップ』，CQ出版(株)
3)『PCIハードウェアとソフトウェア』第4版，インフォクリエイツ

いくら・まさみ　来栖川電工有限会社

第4章

VGA解像度で32ビットフルカラーのフレームバッファ

グラフィックスボードの設計/製作

井倉将実

パソコンと呼ぶからには画面表示も必須だろう．まず，640×480ドットのVGAがどのようなタイミングで画面を表示しているかを解説する．そして，ビデオメモリの設計法から垂直同期/水平同期信号の作成，SDRAMのバースト転送を活用したピクセルデータの読み出し制御などについて解説する．さらに，ピクセルレートとバスの帯域についても考察する．
（編集部）

はじめに

● PC/AT用のVGAカードは使えない？

今回のシステムはPCIバスを装備しているので，手っ取り早く画像表示を行いたければ，PC/ATのビデオカードを使えればいいという意見も出てきそうです．しかしPC/AT用のVGAカードは，非PC/ATプラットホーム上ではそう簡単には使うことができません．

ATアーキテクチャでは，VGAとはあくまで拡張ボードの一種であり，電源投入時はVGA互換レジスタさえもI/O空間にマッピングされていません．VGA互換レジスタをI/O空間にマッピングするには，VGAコントローラの種類によりその初期化方法が異なります．

PC/AT互換機でカードを差し込むだけで画面が表示されるのは，VGAカードにVGA BIOSが搭載されており，VGA BIOS内でそのVGAコントローラに合わせた初期化が行われるからです．よって，VGA BIOSを実行できない非PC/ATプラットホームでは，その初期化を自前で実行しなければならず，汎用的な初期化ルーチンを実現しにくいという事情があるのです．

筆者のところでもこの問題に直面し，一時は「8086エミュレータを作って初期化ルーチンだけ実行するか！」という話も出たほどです．

さらに，2Dアクセラレーションはもちろん，最新の3Dアクセラレーション機能を活用するには，コントローラに対して専用のアクセラレーションコマンドなどを発行しなければなりませんが，これらの仕様が一般には公開されていません．

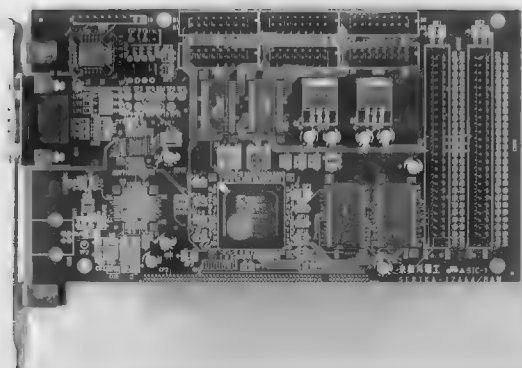
そして，今回のシステムで採用しなかったもっとも大きな理由は，「アーキテクチャが美しくない」という，その一言につきます．

● 組み込み分野でも画像表示の要求が高まる

CPUボードにSDRAMやPCIコントローラをつけたワンボードマイコンの形態のものは，非常に多くの種類が販売されています．SH-4だけに限定した場合でも，本誌を見れば数社程度の広告を見つけることができます．これらのCPUボードは，シリアルポートやEthernetが実装されている場合がほとんどで，画像表示が可能なものはあまりありません．

今後は組み込み機器にも高度なGUIが求められるようになると思われるので，画像表示機能の要求は，今後ますます高まってくると予想されます．

本章ではビデオ表示の基礎知識と，FPGAによる640×480ドットのVGAフレームバッファコントローラの設計方法について解説します．**写真1**に，設計したグラフィックスボードの外観を示します．

〔写真1〕VGA/32ビットフルカラー対応グラフィックスボードの外観

1 画面表示の基礎

最近ではDVI端子と呼ばれる，グラフィックスボードとディスプレイの間をディジタル信号で伝送する新しいインターフェースも登場していますが，ここではもっとも一般的な，アナログRGBでの画面表示について解説します．

● **画面表示の原理**

CRTや液晶ディスプレイなど，現在一般的に使われている画像表示装置は，ラスタスキャンと呼ばれる表示方式を採用しています．

ラスタスキャン方式とは，ディスプレイの左上を始点として，右方向に走査線を走らせ，右端まで走査が完了したら左端の1段低い位置に戻り，再度右端に向かって走査して，これを繰り返し右下まで行います．右下に着いたらまた左上にもどり，以降この操作を延々と繰り返します．

CRTディスプレイの場合，この走査中に，画面上に並ぶ赤(R)，緑(G)，青(B)の蛍光体を発光させ，光の三原色で色を表示させていくわけです．これを高速に繰り返すことで，人間の目には水平/垂直方向に広がる1枚の絵が表示されているように見えるのです(**図1**)．

よって，ディスプレイへの信号として，基本的には水平/垂直同期信号とRGBの映像信号が必要であることがわかります．

● **水平同期信号と垂直同期信号**

画面を左上から右下まで走査しつつ画素を描画するには，CRT側はどのタイミングで左端から右端に移動すればいいのか，また右下から左上に復帰すればいいのかを知らねばなりません．

このタイミングを知らせるための信号が，水平同期信号と垂直同期信号です．また，垂直同期信号は右下から左上に帰ることを指示するので，垂直帰線信号，または垂直帰線同期信号と呼ぶ場合もあります．さらに水平同期信号はHSYNC，垂直同期信号はVSYNCと呼ぶ場合もあります．HSYNC/VSYNCはともにTTL-5Vレベルの信号です．

HSYNCが“L”レベルの期間は，走査線は画面左端に移動します．そして“L”から“H”レベルになると左端から右方向へ走査を開始します．そして横方向表示画素数分のある一定時間が経過したところで再度“L”レベルになるので，走査線を左端に戻して次のラインの走査を開始します．

HSYNC信号が“H”になっている期間は，画素表示を行う期間になります．この期間の間に，画素データをディスプレイ側に出力して画面上に画素の点を，そしてそれが走査されることで横一列につながった線が表示されます．

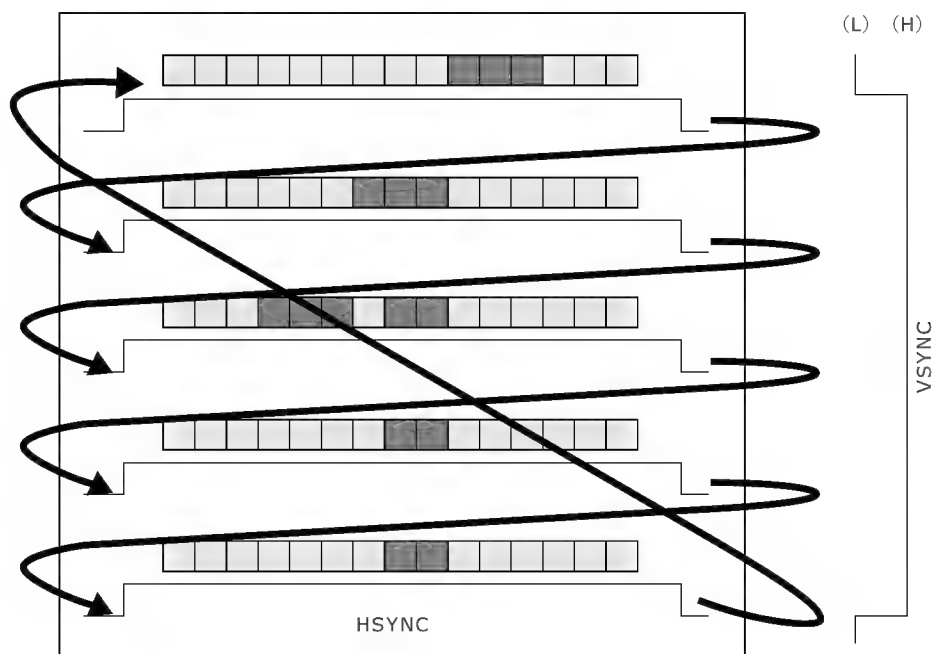
VSYNC信号も同様に，この信号が“L”レベルであると走査線は最上段に移り，“H”レベルになってから垂直方向にHSYNCの回数をカウントするようにします．そして表示ライン数分だけの時間が経過したところで“L”レベルになるので，走査線はまた最上段に移動します．

このように，HSYNCとVSYNCの組み合わせで画面は表示され，基本的にはHSYNCとVSYNCの両信号が“H”レベルの矩形領域が「絵が表示される画面」となるのです．また，1秒間に何回画面表示が行われるかをリフレッシュレートと呼び，フレーム/秒(以下fps)という単位で表します．

● **ブランク処理**

たとえばCRTで，画面の上下や左右の端いっぱいまで画面を表示させると，端の表示が歪んで表示される場合があると思います．**図1**で走査線が右から左に戻る瞬間(矢印が曲がる部分)は，どうしても表示が安定しません．そこで走査の初めと終わりに，走査が安定するまで少し余裕をもたせます．これをブランク処理と呼びます．

さらに走査線が下から上に戻るときは，水平方向以上に表示が安定しません．よってVSYNCの最初と最後には，1ラインまるごと表示させずに水平走査のみ行うブランク処理を数ライン分入れます．

〔図1〕ラスタスキャンによる画面表示の原理

〔表1〕各解像度/リフレッシュレートでの各表示タイミング

画面解像度	ドット	640 × 480	640 × 480	800 × 600	800 × 600	1024 × 768	1024 × 768	1280 × 1024	1280 × 1024	1600 × 1200	1600 × 1200
通称解像度	—	VGA	VGA	SVGA	SVGA	XGA	XGA	SXGA	SXGA	UXGA	UXGA
リフレッシュレート	fps	60	75	60	75	60	75	60	75	60	75
ピクセルクロック	MHz	23.856	30.722	28.216	48.906	64.109	81.804	108.883	138.542	160.963	205.993
水平同期周波数	kHz	29.820	37.650	37.320	47.025	47.700	60.150	63.600	80.175	74.520	93.975
水平同期間隔	Pixel	800	816	1024	1040	1344	1360	1712	1728	2160	2192
水平フロントポーチ幅①	× 8pixel	2	3	4	5	7	7	10	11	13	15
水平表示期間幅②	× 8pixel	80	80	100	100	128	128	160	160	200	200
水平バックポーチ幅③	× 8pixel	10	11	14	15	20	21	27	28	35	37
水平同期パルス幅④	× 8pixel	8	8	10	10	13	14	17	17	22	22
垂直同期帰線間隔	HSYNC-Line	497	502	622	627	795	802	1060	1069	1242	1253
垂直フロントポート幅⑤	HSYNC-Line	1	1	1	1	1	1	1	1	1	1
垂直表示期間幅⑥	HSYNC-Line	480	480	600	600	768	768	1024	1024	1200	1200
垂直バックポーチ幅⑦	HSYNC-Line	13	18	18	23	23	30	32	41	38	49
垂直同期帰線パルス幅⑧	HSYNC-Line	3	3	3	3	3	3	3	3	3	3

これらのブランク信号の制御で，HSYNC/VSYNCの各信号が"L"から"H"レベルに変化するタイミングでアサートするブランク信号期間をフロントポーチ，そして後半の"H"から"L"に変化するタイミングでアサートするブランク信号期間をバックポーチと呼びます．

ブランク処理は，BLANK*信号がアサートされている期間，その画素を黒で置き換えます．当然，HSYNCやVSYNCの"L"レベルの期間も，走査線が戻る途中なのでBLANK*は有効となります．

● **実際の信号の各タイミング**

PC/AT互換機におけるビデオ関連の規格は，VESAにより規定があります．ここではより具体的に，640 × 480ドットのVGA解像度，60fpsの場合の各タイミングの時間を見てみましょう．

まず，1秒間に60フレーム画面を表示するということは，VSYNCは60回の"L"と"H"を繰り返すことになります．つまり，VSYNCは16.667msごとに"L"レベルになるわけです．

次にHSYNCですが，VGAなので表示するライン数は480回になります．さらにVSYNC方向のブランク時間は，VESA規格によると17HSYNC期間と規定されています．よって合計497回，HSYNCが"L"レベルになることになります．16.6msで表示される1画面中に，497回のHSYNCが存在するわけですから，HSYNCは33.53μsごとに"L"と"H"を繰り返します．

さらに，HSYNCの"L"レベルの期間はVESA規格で80画素分と規定されており，さらに80画素分の表示空白期間(ブランクマージン)を持たせるようにも決められています．つまり1HSYNCは，640 + 80 + 80画素で合計800画素となります．

以上から1画素あたりの表示時間は33.53μs/800画素＝41.9ns，逆数を取ると23.856MHzとなり，これをピクセルクロックと呼びます．

表1に，VGAからUXGAまでの各解像度で，さらにリフレッシュレートを60fpsと75fpsとしたときの各タイミングを示します．また，それぞれのパラメータの該当部分を**図2**に示します．

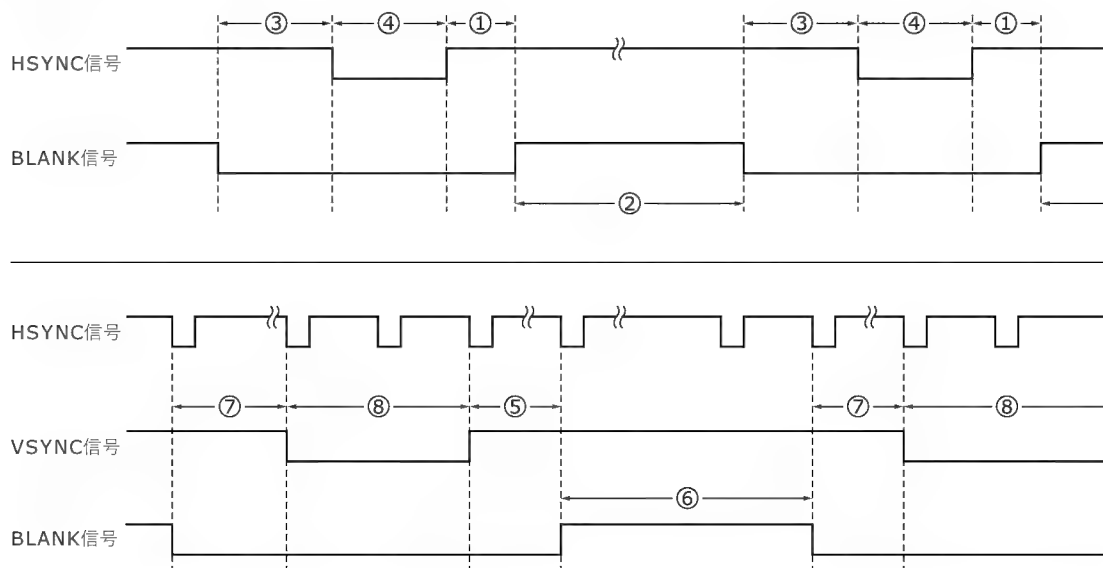
〔図2〕各タイミングパラメータの関係

● タイミングと発色数の関係

表1をよく見ると，表示発色数のパラメータがありません．アナログRGB接続の場合，ディスプレイに出力するRGBの各信号を何段階で制御するかが発色数に関わってきます．RGBを各8ビット分解能で制御すれば24ビットフルカラーとなり，各5ビットであれば15ビットで32768色表示となります．

2 グラフィックスコントローラの仕様検討

8ビットパソコン時代には，ビデオメモリに文字コードを書き込むだけで，ビデオコントローラが文字コードに該当するフォントデータから，それをラスタに変換して画面に文字を表示するテキストVRAMという方式もありました．この方式は少ないデータ量でテキストを表示できますが，グラフィックスを表示することはできません．

逆にDOS/Vの日本語テキスト画面は，ハードウェア的にはグラフィックスモードでありながら，ソフトウェアでフォントデータから文字を描画して表示しています．つまり，十分な解像度のグラフィックスを描画できる能力があれば，テキスト画面モードを再現することができるのです．

ここではグラフィックス画面を表示できるグラフィックスボードを実現します．

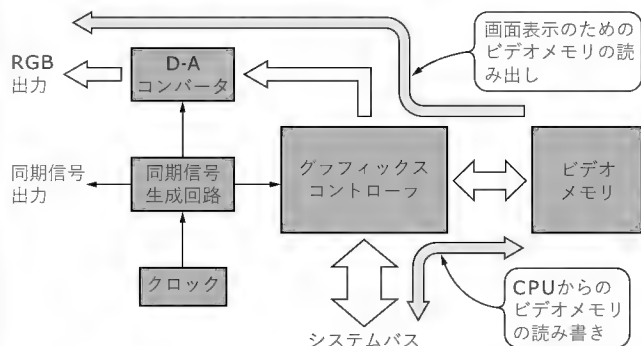
● 基本的なグラフィックスボードの構成

図3に，画面表示に必要な機能を示します．

まず，表示する画像データを保持するビデオメモリが必要です．また，今回はアナログRGB対応を考えているので，RGBの各信号はアナログ電圧で出力する必要があります．よって出力段にはD-Aコンバータが必要です．また先ほど説明した各種タイミングを生成する回路も必要でしょう．そしてこれらをシステムに接続するための，ISAバスやPCIバスといった何らかのシステムバスも必須です．

グラフィックスコントローラとは，これらを一つにまとめるための制御部分と考えることができ，今回はこれをFPGAで設計するわけです．

チラツキのない表示を行うには，ビデオメモリからD-Aコンバータに絶え間なくデータを送り続ける必要があります．当然，

〔図3〕基本的なグラフィックスボードの構成

表示するデータを書き替えるために，CPUからビデオメモリへの書き込み動作も必要ですし，逆にビデオメモリに書き込んだデータをCPUが読み出すこともあるかもしれません(書き込み専用ビデオメモリというのも悪くないが，あまり一般的ではない)．さらにビデオメモリとしてDRAM系のデバイスを採用した場合，リフレッシュも考えなければなりません．

● ビデオメモリの選択

図3でわかるように，ビデオメモリへのアクセスは画面表示のための読み出しと，CPUからの読み書きの2系統があります．また，そのアクセスは，画面表示は画素の走査と同様にシリアルアクセスしか発生しませんが，CPUからの読み書きはランダムアクセスができなければなりません．

そのため以前は，画面表示側にはシリアル読み出し専用のポートを，CPU側にはランダムアクセスで読み書き可能なポートを用意した，2ポート構成のいわゆるVRAMと呼ばれるメモリが使われていました．そしてパソコンのメインメモリの主流がDRAMからSDRAMに移行したように，VRAMにもSGRAMと呼ばれるメモリが登場しました．

しかし，これらのビデオRAMは，メインメモリとして大量に使われるDRAMやSDRAMと比較して価格が高いことに合わせ，近年のPC周辺機器の低価格化のあおりを受けたPC/AT用VGAカードの低価格化で，最近ではほとんど使われなくなってしまいました．

こうなると，ますますビデオメモリ用のメモリを製造するベンダは減り，入手も困難になるという悪循環に陥っています．

● 1画面の表示に必要なビデオメモリの容量

解像度が低く，表示発色数も少なければ，ビデオメモリの容量は少なくて済みます．たとえばVGA解像度では，640 × 480ドット＝307200ドットというピクセル数になります．発色数が最大256色であれば，1ピクセルあたり8ビットで表示できるので，そのまま必要なビデオメモリの容量は307200バイトとなります．フルカラーであれば24ビットなので，その3倍の921600バイト，約1Mバイトとなります．同様にXGAでフルカラーであれば，約2.3Mバイトになります．

さて，フルカラー表示は1ピクセル24ビットで表現できますが，24ビットつまり3バイトという値は少々中途半端です．2のn乗倍にならないと座標計算などが面倒になります．そこで上位に8ビットを追加して1ピクセル32ビットすると非常にスマートになります．追加した8ビットは単に情報を保持するために使ったり，画面合成するときの何らかのパラメータとして使うなど，応用しだいで使いこなすことが可能になります．

● ビデオアーキテクチャの決定

これをいかに決定するかが，グラフィックスコントローラを設計するうえでもっとも楽しい(?)作業かもしれません．

たとえば，2画面分のビデオメモリと切り替え回路を実装して，画面の描画が完了したら画面を切り替えることでチラツキをなくすという方法，またその2画面を，片方の画面を背景に，

もう片方の画面の一部を透過させて重ね合わせ表示可能にする方法，さらには表示する解像度より広いビデオメモリを実装し，その広い座標空間内の任意の位置から画面に表示を開始するという方法など，説明を始めたらきりがありません．アイデアは無限です．

今回筆者は，次のように基本的なアーキテクチャを決定しました(**図4**)．

- 2048 × 2048 ドットの仮想画面中の任意の 640 × 480 ドットを表示
- 1 ピクセルの構成は上位ビットから α，R，G，B の各 8 ビット
- 発色モードはフルカラー表示固定

以上から，16M バイトのビデオ容量が必要なことになります．16M バイトの容量を実現するとなると，現状では SDRAM を選択するのが無難なところでしょう．

● バス帯域の検討

グラフィックスコントローラの設計でもう一つ重要な決定事項が，次に説明するバス帯域の検討です．

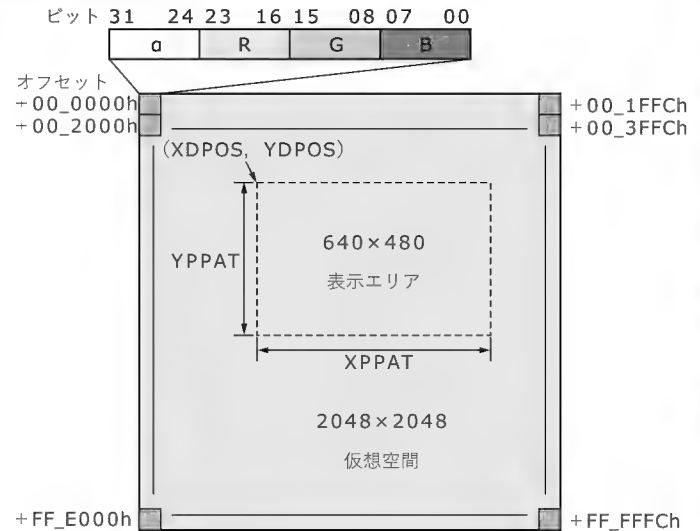
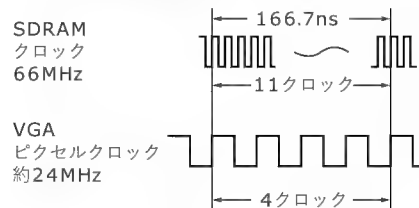
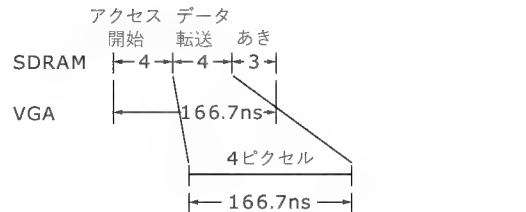
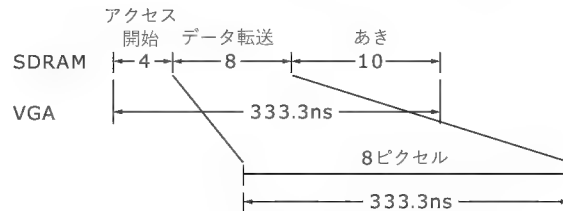
今回採用を考えているビデオメモリは SDRAM です．通常の SDRAM はポートが一つしかありません．つまり，画面表示のための読み出しと，CPU からの読み書きで同じポートを使わなければなりません．そして SDRAM も DRAM の一種ですから，定期的にリフレッシュも必要になります．つまり一つのポートを三つの用途で使うことになるわけです．

ここで仮に，32 ビット幅の SDRAM を 66MHz のクロックで動かすとしましょう．SDRAM のアクセスには RAS/CAS などの制御が必要で，データが出てくるまでに 4 クロックかかります．また，VGA 表示時のピクセルクロックを近似して 24MHz とします．すると 66：24 なので，SDRAM 側の 11 クロック時間で，画面に 4 ピクセル表示される計算になります〔**図5(a)**〕．

SDRAM はアクセス開始に 4 クロックかかるとすると，シングル転送で 1 ワードしか読み出さなくても，合計 5 クロックの時間がかかるわけです．VGA で 4 ピクセル表示する時間に，SDRAM をシングル転送させると，なんと 2 回しか転送を実行できません．4 ピクセル分のデータが欲しいのに 2 ピクセル分しか読み出せないことになり，これは画面表示のための SDRAM の読み出しが，実際の画面表示にまったく追いついていないことを意味します．

グラフィックスで SDRAM を使う場合は，必然的に数ワード連続してデータを読み出すバースト転送が必須になります．

では，もっとも基本的な 4 ワードバースト動作時で考えます．SDRAM は 11 クロック中，アクセス開始に 4 クロック，データの読み出しに 4 クロックで，4 ピクセル分のデータを読み出すに 8 クロックかかるわけです．する

〔**図4**〕今回のグラフィックスコントローラの基本仕様

〔**図5**〕バス帯域の検討

(**a**) SDRAMクロックとVGAピクセルクロックのクロック比

(**b**) 4ワードバースト/4ピクセル

(**c**) 8ワードバースト/8ピクセル

(**d**) 8ワードバースト/16ピクセル

と，あと3クロック分の時間しか残りがありません．この3クロックの時間を，CPUからのアクセスとリフレッシュで使うことになります〔**図5(b)**〕．

実際には3クロックではCPUからSDRAMへアクセスすることはできないので，倍の8ピクセル，さらに倍の16ピクセル分を一気に読み出します．たとえば8ピクセル分なら，SDRAM側は22クロック分の時間になるので，22－(アクセス開始4クロック＋データ8クロック)＝10クロックの空き時間になります〔**図5(c)**〕．

この例では，SDRAMを1ワードアクセスして32ビット，つまり1ピクセル分のデータを読み出していますが，SDRAMのデータバス幅を倍の64ビットにするとどうでしょうか．アクセス開始の4クロックは同じですが，その次からは1ワードで2ピクセル分のデータを読み出せるわけです．つまり，**図5(d)**のように8ワード分読み出すと，16ピクセル分のデータを読み出すことが可能になり，32クロックもの空き時間を確保できます．CPUからのアクセスにも十分時間を確保することができるわけです．

実際にはSDRAMのアクセスには，さらにプリチャージ時間などのクロックも必要なので，これよりも空き時間が減ります．SDRAMをどんなモードでどう制御するかも考慮に入れた計算が必要です．

また，当然のことながら，SDRAMからの読み出しとピクセルクロックの表示出力は速度差があるので，FIFOなどのバッファを置いて，いったんそこに詰め込んでおく必要があります．

● 最終的なグラフィックス構成の決定

2ポートのビデオメモリが使えれば，いかに余裕のある設計ができるかがわかるかと思います．しかし，入手困難なメモリは採用できません．

SDRAMを使う場合は，画面表示のための読み出し時間を減らし，少しでもCPUアクセスのための時間を広げるよう，あれこれ構成を考える必要があります．そのグラフィックスボードで動画を表示させようものなら，CPUアクセスにさらに広い帯域が必要になるのは目に見えています．

以上の点を考慮し，今回は32ビット幅のSDRAMを2個並べて，グラフィックスコントローラとビデオメモリの間を64ビット幅で接続することにします．

3 グラフィックスボードの設計

今回設計するグラフィックスボードのブロック図を**図6**に示します．以下，各部について説明します．

● PCIバスインターフェース

今回のボードはPCIバス上に実装するので，システムバスとのインターフェースにはPCIバスを使用します．CPUからの描画は，すべてPCIバス経由でのアクセスになるわけです．もちろんCPU以外のバスマスタが，直接ビデオメモリに転送をしてくる場合もあります．

今回のVGA/フルカラーモードで60fpsの非圧縮動画転送を行った場合，その転送帯域はおよそ74Mバイト/秒になります．PCIバスの最大転送帯域が133Mバイト/秒なので，55%の転送帯域が単なるビデオデータの転送に使われます．このCPUインターフェースには，十分に高速なデータ転送に耐えるだけの性能が要求されます．

実際の動画再生ソフトウェアでは，限りある転送帯域を有効

〔**図6**〕**設計したグラフィックスボードのブロック図**

に活用するため，データを1/10程度まで圧縮して転送量を小さくしていますが，高速で動かせればそれに越したことはありません．

● **ビデオメモリ（SDRAM）コントローラ**

今回のグラフィックスボードはビデオメモリとしてSDRAMを採用したので，ビデオメモリコントローラはすなわちSDRAMコントローラとなります．

ビデオメモリを制御するSDRAMコントローラは，第2章で紹介したSH-4のメインメモリ用SDRAMコントローラよりも，複雑でかつ転送帯域を保証しなければならない回路です．

もしSDRAM読み出しが画面出力に間に合わなかった場合，画素読み出しが間に合わないということは，結果として画面が正しく表示されなくなります．具体的には激しいチラツキや崩れた表示になります．

● **ビデオメモリ**

ディスプレイに表示する画素データを保持するメモリです．今回は2048×2048ドット/1ピクセル32ビットという仕様から，全容量は16Mバイトとなります．また，転送帯域を確保するため，32ビット幅の64MビットSDRAMを2個並べ，データバス幅を64ビット構成にしています．

このように，実際に画面に表示される範囲よりも広いビデオメモリを実装することで，いくつかの利点が生まれます．

たとえば，表示開始位置をパラメータで変更可能な構造にして，表示位置を少しずつ移動させることで，実際のビデオメモリ上のデータをコピー転送で移動させなくても，画面表示を移動させることができます．いわゆるハードウェアスクロールを実現できるわけです．

また，表示画面の数画面を同時に確保できるので，ある部分を表示中に非表示部部分を書き替えることで，画面のチラツキきを抑えることができます．表示範囲と同じ位置にCPUからの書き込み動作を行うと，どうしてもチラツキが発生してしまいます．これを回避するには，2画面，3画面分の領域を確保し，表示中の範囲は書き替えずに，表示していない画面エリアを書き替え，書き替えが完了したらその表示をその範囲に切り替えるわけです．このようなテクニックをダブルバッファ，トリプルバッファと呼ぶこともあります．

なお，この切り替えるタイミングを判定する方法として，ホストCPUが何らかの方法で画面の表示時間を計測するなどといった方法ではなく，グラフィックスコントローラが画面表示の垂直同期帰線時間になるタイミングを判定して，それに同期して切り替えると，画面の切り替え表示がスムースになります．垂直同期帰線時間のタイミングは，ステータスレジスタをポーリングすることでも取得できますが，今回のコントローラでは，PCIバスに割り込みとして出力することもでき，ホスト側が割り込み処理で画面表示位置を切り替えることも可能です．

● **ビデオタイミング生成回路**

ビデオタイミング生成回路の役割は，HSYNCとVSYNC，そしてBLANK*信号をタイミングチャートにしたがって生成することです．このタイミング生成回路の基本となるクロックは，1画素の表示クロックつまりピクセルクロックであり，このクロックを数えることで，各タイミング信号の“H”パルス幅や“L”パルス幅を決めます．

ただしタイミングを作るにあたって，ピクセルクロックをそのままカウンタに使うことはしません．今回の仕様のようにVGA程度のピクセルクロック（23.856MHz）であれば問題なくカウンタも動くでしょうが，解像度やリフレッシュレートが上がり，ピクセルレートがどんどん高速になると，速度が速すぎてカウンタが回らないこともあるからです．またすべてのタイミングを，1ピクセル単位で表示位置を調整するほどの精度にする必要がありません．

そこで，一般的には基準となるクロックを8ピクセルごとに1回カウントするように，8：1プリスケーラを用意しておきます．ここではこの信号をDIV8クロックと呼びます．

たとえばVGA/60fpsのタイミング生成回路を作る場合を想定しましょう．HSYNC信号を作るには，四つのカウンタを使用します．

- 水平フロントポーチ幅
- 水平表示期間幅
- 水平バックポーチ幅
- 水平同期パルス幅

この四つのカウンタによって算出されたHSYNC信号の“L”と“H”を繰り返す周期が，800ピクセルクロック分に相当します．

そしてこのうち，水平同期パルス幅期間をカウントしている間はHSYNC出力が“L”レベルとなり，それ以外の期間は“H”レベルです．フロントポートカウンタとバックポーチカウンタの値は，ブランク信号を作るために必要なラインなので注意してください．

同様に，次はVSYNC信号側を考えましょう．

こちらもHSYNCと同様に，次の四つのカウンタが必要になります．

- 垂直フロントポート幅
- 垂直表示期間幅
- 垂直バックポーチ幅
- 垂直同期帰線パルス幅

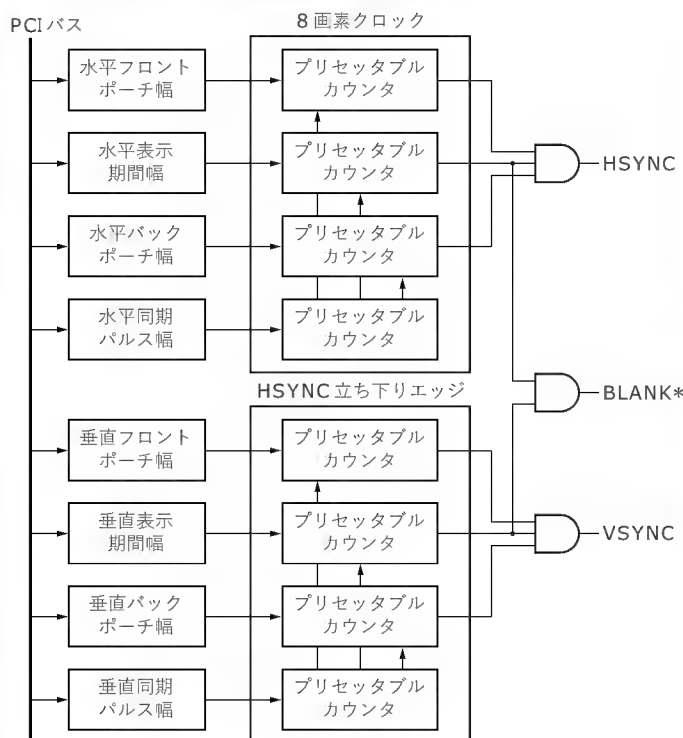
こちらのカウンタで数える元になるのは，HSYNC信号の回数です．つまり表示ライン数を数えています．

ブランク信号は合計八つのカウンタのうち，表示期間時のみ“H”レベルにし，それ以外は“L”レベルにするという回路です．これで表示期間以外の間はブランク信号が有効になり，ディスプレイ側は画面上に余分な画素を表示したりはしません．

これらをまとめたタイミング回路のブロックを**図7**に示します．

● **ピクセルデータバッファ**

ビデオタイミング生成回路の要求によりビデオメモリから読み出された画素データを，後段のD-Aコンバータに対して送り

〔図7〕ビデオタイミング生成回路のブロック図

出す回路です．この回路も，ビデオタイミング生成回路から出力されるHSYNCやVSYNC，BLANK*信号によって送り出しを制御します．

また，この部分に実装する機能として，強制的に画面表示をオフにする機能や，画面表示の輝度を制御するフェイドコントロール機能があります．フェイドコントロールとは，ビデオメモリから読み出したRGBの各表示レベルを除算し，輝度を調整して出力します．この輝度調整を画面表示のタイミングと合わせて可変することで，真っ黒の画面から映像が浮かび上がってくる(フェイドイン)，または表示画面が徐々に暗くなる(フェイドアウト)を実現できます．

今回設計する回路は，ピクセルクロックが約24MHz程度なので，現在のFPGAではそれほどたいへんな回路にはなりません．しかし**表1**でわかるように，解像度が上がれば上がるほど，リフレッシュレートが上がれば上がるほど，ピクセルクロックは高くなります．解像度によっては100MHzを超えるクロックを要求される部分なので，筆者はいつも設計に気をつけています．

● メモリマップとレジスタ構成

以上，設計したグラフィックスボードのPCIデバイスのコンフィグレーションレジスタを**表2**に，CRTコントロールレジスタ群のレジスタ一覧を**表3**に示します．今回の仕様ではVGA固定となっているので，各レジスタ中のパラメータでは，VGAで意味をもつモードしか動作しない点に注意してください．

〔表2〕PCIデバイスのコンフィグレーションレジスタ一覧

ベンダID	0x6809
デバイスID	0x8114
クラスコード	0x03 0x80 0x00 ディスプレイコントロールクラス/ その他のディスプレイコントローラ
ヘッダタイプ	0x00
ベースアドレス0	1Mバイトのメモリ空間 CRTコントロールレジスタ群を配置
ベースアドレス1	16Mバイトのメモリ空間 ビデオメモリをリニアにマッピング
割り込み	INTA#使用

4 帯域確保のテクニック

画面表示でいちばん注意しなければならないのは，画面表示を止めてはならないということです．たとえば，HDDなどではときどきウェイトが入る場合があっても，少しパフォーマンスが下がる程度で実害はありませんが，画面表示がちらつくグラフィックスボードは，はっきりいって使い物になりません．

たとえば，VGA/60fpsモードのピクセルクロックは約24MHzです．そして必ず24MHz = 41.6ns間隔で画素出力を行わなければなりません．第2章でSDRAMの使い方を解説しましたが，100MHzで32ビットバス幅帯域をもつSDRAMコントローラから比べれば，これはそれほど難しいようには思えないかもしれません．

しかし，必ず41.6ns間隔で画素出力を行わななければならないというのは，リフレッシュやプリチャージ，またCPUからのアクセスもすべて包括してなおかつ，この速度を出さねばならないということです．

画面表示のためのSDRAMアクセスをしつつ，プリチャージ，リフレッシュ，CPUアクセスの時間を確保するための方法を，いくつか説明します．

● バースト転送を活用する

第2章のSDRAMコントローラの章でも説明したように，SDRAMの読み出しは1ワード目のデータが出てくるまでは時間がかかり，2ワード目以降の読み出しは連続的に行うことができます．2ワード目以降そのまま連続して読み出せるワード数は，通常のSDRAMでは8ワードまで可能です．すでに転送帯域の検討で説明〔**図5(b)**，**(c)**〕したように，SDRAMのバースト転送を活用することで，バス帯域幅を確保することができます．

さらにSDRAMメモリは，同一RASアドレス/バンク内では，連続してCASアドレスを発行し，継続してデータを読むことができます．たとえば，今回のグラフィックスボードでも採用している，筆者が好んで使っているSDRAM MT48LC2M32(マイクロン)では，512ワード分の連続空間はバンクアクティブを行わなくても連続に読み出すことができます．

512ワードを連続で読み出すには，まずはじめにRASにより

〔表3〕CRTコントロールレジスタ一覧

▶ HSYNC：水平同期タイミング関連レジスタ

オフセット	名 称	意 味
+00h	HSPW	ビット5〜0：HSYNC/水平同期パルス幅設定レジスタ
		ビット15：HSYNCパルス極性反転ビット
+04h	HFP	HSYNC/表示開始位置マスク幅設定レジスタ
+08h	HV	HSYNC/水平方向表示期間幅設定レジスタ
+0Ch	HBP	HSYNC/バックポーチ幅設定レジスタ

▶ VSYNC：垂直同期タイミング関連レジスタ

オフセット	名 称	意 味
+10h	VSPW	ビット5〜0：VSYNC/垂直同期パルス幅設定レジスタ
		ビット15：VSYNCパルス極性反転ビット
+14h	VV	VSYNC/垂直方向表示期間幅設定レジスタ

▶ HSYNC/VSYNC割り込み制御関連レジスタ

オフセット	名 称	意 味
+18h	HIP	ビット31：HSYNC期間中
		ビット15：HSYNC割り込みステータスビット
		ビット14：HSYNC割り込みイネーブルビット
		ビット10〜0：水平同期(HSYNC)割り込み位置設定レジスタ
+1Ch	VIP	ビット31：VSYNC期間中
		ビット15：VSYNC割り込みステータスビット
		ビット14：VSYNC割り込みイネーブルビット
		ビット8〜0：垂直帰線同期(VSYNC)割り込み位置設定レジスタ

▶表示サイズ/開始位置関連レジスタ

オフセット	名 称	意 味
+20h	HPLAT	水平方向表示位置微調整レジスタ
+24h	HPPER	水平方向表示サイズ設定レジスタ
+28h	VPLAT	垂直方向表示位置微調整レジスタ
+2Ch	VPPER	垂直方向表示サイズ設定レジスタ

注：オフセットはベースアドレスレジスタ0のアドレスを先頭アドレスとしたとき

▶水平/垂直方向表示開始ピクセル位置設定レジスタ

オフセット	名 称	意 味
+30h	XDPOS	水平方向表示開始ピクセル位置設定レジスタ
+34h	YDPOS	垂直方向表示開始ピクセル位置設定レジスタ

▶ CRTコントロール関連レジスタ

オフセット	名 称	意 味
+40h	DISPCTRL	CRTコントロールレジスタ群
		ビット15：DISPLAYコントロール
		“1”：表示オン
		“0”：表示オフ
		ビット14：HSYNC/VSYNCコントロール
		“1”：イネーブル
		“0”：ディセーブル
		ビット13〜12：ピクセルクロックセット
		“11”：57.272MHz(予約)
		“10”：28.636MHz(予約)
		“01”：57.272MHz (XGAピクセルクロック)(予約)
		“00”：28.636MHz (VGAピクセルクロック/デフォルト)
		ビット11：ピクセルカラーモードセレクト
		“1”：32ビットカラー
		“0”：16ビットカラー(予約)
		ビット7〜0：VESAローカルCRTコントローラ係数レジスタ
		“10h”：640×480/ リフレッシュレート60Hz
		“12h”：640×480/ リフレッシュレート75Hz(予約)
		“20h”：800×600/ リフレッシュレート60Hz(予約)
		“22h”：800×600/ リフレッシュレート75Hz(予約)
		“30h”：1024×768/ リフレッシュレート60Hz(予約)

バンクアクティブし，t_{RCD}後にCASを発行します．ここからCL値経過後にデータが8ワード分連続に出てきますが，ちょうどCASを発行してから8クロック後に次のCASアドレスを入れると，1回目のCASによる8ワードデータ出力に連続して，つぎの8ワードデータが出力されます．

つまり，SDRAMコントローラのシーケンサを8クロックごとにCASを発行するように設計すると，このMT48LC2M32や同等品に関しては，最大512ワードまで連続してデータを読み出すことができるので，SDRAMの読み出しあたりのオーバヘッドは見かけの上では限りなくゼロになります．

このテクニックを使ったSDRAMコントローラのステートマシンの一部を**リスト1**(章末)に示します．

● **FIFOバッファを深く**

FIFOを使うことの利点は，書き込み側のバスクロックと読み出し側のバスクロック差を考えなくてすむということです．SDRAMから読み出したデータをいったんFIFOに書き込む部分は，SDRAMのクロック速度で動作させます．実際に画面に出力するためのFIFOから読み出す部分は，ピクセルクロックと同期したクロックで動作させます．ピクセルクロックで読み出すといっても，すでに説明したブランキング時は画面表示をしないので，ビデオタイミング生成回路のBLANK信号にしたがって読み出しを制御します．

このようにSDRAMの速度とピクセルデータ読み出しの速度を非同期に設計することで，ピクセル読み出し速度の異なるさまざまな画面モードにも対応することも可能になります．

さて，バースト転送により数十ピクセル分のデータを読み出しても，今度は読み出したデータを実際に表示されるタイミングまで，どこかに保存しておかなければなりません．バーストワード数を多くすると，それだけ短時間に読み出すワード数が多くなるので，結果，FIFOバッファを深くする必要があるわけです．

FIFOの段数は，メモリコントローラの速度とピクセルレート，それにメモリの全転送帯域中に，CPUからのアクセスの割

合をどれくらいにするのかという点を考慮して，さまざまに変わると思います．たとえば，全転送帯域の1/3を画面出力に割り当ててもいいなら，FIFOは32ワードもあればいいでしょう．

画面出力用の転送帯域が少なくなればなるほど，CPUからのアクセスにも応答しやすくなるということで，頻繁に画面書き換えを行う用途や動画表示を考えているのであれば，FIFOを深くして一度に大量のピクセルデータを取り込むような工夫をします．

● ビデオメモリのビット幅を増やす

さらに転送帯域を確保したい場合は，転送帯域の検討の**図5(d)**の方法である，32ビット幅よりも64ビット幅，さらに128ビット幅というように，バス幅を広げて転送帯域を確保する方法も非常に有効です．1ワードの転送で2ピクセル，4ピクセルを同時に読み出せるので，画面表示のためにSDRAMにアクセスする時間が相対的に少なくなります．バス幅が倍になれば，ほぼ倍の帯域を確保できると考えられます．

しかし，データバス幅を広げる方法は，グラフィックコントローラとビデオメモリの間の信号線を増やさなければなりません．データバス幅が広がれば広がるほど，データバスの信号線を基板上で配線するのが困難になってきます．

● クロックを上げる

転送帯域を確保するためには，SDRAMの動作クロック周波数を上げるという方法もあります．もちろん，FPGAやSDRAMがその速度に対応できれば，という前提になります．100MHzのバスクロックを133MHzに上げるだけで，33%高速に転送できます．さらに，画面表示のための読み出し動作は，読み出しワード数が同じならかかる時間は短くなるので，実質2倍ぐらいの性能向上が期待できます．

まとめ

ここではグラフィックスコントローラに要求される機能をひととおり述べましたが，VGAクラスであっても，設計段階でこれぐらいの機能に対する考慮をしておくと，今後，高解像度，動画転送，複数画面の重ね合わせ処理などの要求にも対応できることでしょう．

いくら・まさみ　来栖川電工有限会社

〔リスト1〕ビデオメモリ用SDRAMバーストアクセス対応ステートマシンの一部

```
-- Wait for Read data transfer.
when WAIT MEM ACC =>

     case WORD COUNTER(4 downto 0) is
          when "00001" =>
          -- **** DAC side data transaction **** --
               if ( MPU ACC START = '1' ) then
                         -- At current transaction if for MPU.
                    nMCS    <= "00" ; -- Prechage selected bank
                    nMRAS   <= '0' ;
                    nMCAS   <= '1' ;
                    nMWE    <= '0' ;
               else
                    nMCS    <= "11" ; -- No operation command.
                    nMRAS   <= '1' ;
                    nMCAS   <= '1' ;
                    nMWE    <= '1' ;
               end if ;

          when "00010" =>
               nMCS      <= "11" ; -- No operation command.
               nMRAS     <= '1' ;
               nMCAS     <= '1' ;
               nMWE      <= '1' ;

          when "00011" =>
               if ( DAC ACC START = '1' ) then
                    FF WR EN  <= '1' ;
                         -- PIXEL data holding fifo write start !!
               end if ;

          when "00100" =>
               hold SDRAM D I <= "1111" ;
          when "00101" =>
     -- **** At CL = 4 **** --
          -- **** MPU side data transaction **** --
               hold SDRAM D I <= "0000" ;
               READY Port     <= MPU ACC START ;
          -- **** DAC side data transaction **** --
               if ( MPU ACC START = '1' ) then
                         -- At current transaction if for MPU.
                    VRAMC NSTATE := ACCESS COMPLETE ;
               end if ;

          when "00111" =>
               MA ADRS CAS3 <= '1' ;
          --   MA(3)      <= '1' ; -- UPPER 16PIXEL.
          --   MA(10)     <= '1' ;
                         -- READ auto-prechage sign flag.
          when "01000" =>
               nMCS      <= "00"; -- Read/Write command.
               nMRAS     <= '1' ;
               nMCAS     <= '0' ;
               nMWE      <= '1' ;

          when "01001" =>
          --   MA        <= ( others => '0' ) ;
               nMCS      <= "11" ; -- No operation command.
               nMRAS     <= '1' ;
               nMCAS     <= '1' ;
               nMWE      <= '1' ;

          when "01100" =>
               nPIX ACK  <= '0' ; -- Set acknowledge PIX ACK*
          when "01110" =>
               nPIX ACK  <= '1' ; -- Negate acknowledge PIX ACK*

          when "10010" =>
               VRAMC NSTATE := ACCESS COMPLETE ;

          when others =>
               null ;

     end case ;
```

第5章

M16CマイコンとPCIデバイスでキーコードを変換する

PS/2キーボード&マウスインターフェースの設計/製作

山武一朗/藤が丘勝信

コンピュータシステムの入力デバイスとして，もっとも普及しているのがキーボードとマウスであろう．ここではPS/2キーボードおよびPS/2マウスの通信プロトコルについて解説し，M16Cマイコンとの接続部分について解説する．また，PS/2キーボードやマウスのデータをよりスマートなコードに変換してPCIバス側へ出力し，ホストであるSH-4から利用しやすくするよう，PCIバス上にキーボードインターフェース回路を設計する．（編集部）

はじめに

画面表示出力ができたら，次はキーボードやマウスの入力でしょう．キーボートとマウスとくれば，これから採用するシステムなら，インターフェースはUSBを採用すべきという声が出てくるのは当然です．しかしプロローグで説明したように，キーボードパワーONなど，ある意味マニアック(!?)な仕様を実現するために，ここではあえてPS/2インターフェースを採用しました．

ただし，PS/2インターフェースの制御をそのまま直接，ホストCPUであるSH-4にさせるつもりはさらさらありません．せっかくの美しいアーキテクチャ(?)に，PS/2というレガシーなインターフェースを持ち込みたくはありません．くわしくは後述しますが，とくにPS/2キーボードのキーコードの統一性のなさを見れば，この気持ちを理解してもらえると思います．

そこでPS/2デバイスから送られてくるデータをよりスマートなコード体系に変換し，さらにシステムバスであるPCIバスに接続できるように考えてみます．キーコードの変換はそう簡単ではなく，また今回は誌面の関係で解説しませんが，バッテリバックアップ付きのリアルタイムクロックを接続することで，タイマ起動を可能にするなど，さまざまな電源制御/システム制御を行うことも考え，ここにマイコンを採用することにしました．

図1にキーボード&マウスインターフェースの構成を示します．最終的にはATX電源の制御なども考えているので，システムコントローラと呼んでいますが，ここでは誌面の都合から，キーボード&マウスインターフェース部分にしぼって解説します．

今回，マイコンにはM16C(三菱電機)を採用しました．なぜM16Cなのかは諸般の事情(しがらみ?)によります(笑)．日立製作所と三菱電機の半導体部門の事業統合により，今後このマイコンがどうなるのかは不安なところがありますが，M16Cはカーエレクトロニクスの分野でそれなりにシェアのあるマイコンなので，当分の間は消えないと考えています．

写真1に試作したシステムコントローラを示します．なお，PCIボードには次章で解説するATAインターフェースも実装しています．よってPCIデバイスは，マルチファンクションデバイスとなっています．

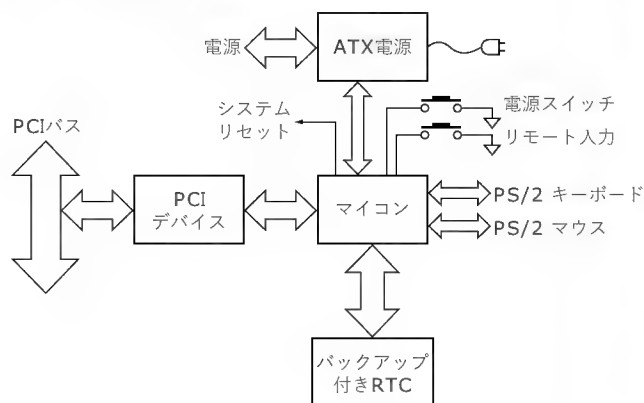
PS/2インターフェースの通信プロトコル

それではまず，PS/2キーボードとマウスの通信プロトコルについて解説します．

- クロック線とデータ線の2本のみ

図2にPS/2コネクタのピン配置を示します．PS/2キーボードもPS/2マウスもピン配置は同じです．このようにPS/2インターフェースは，電源とグラウンドを除くと，CLK線とDATA線の2本しかありません．

これでは片方向通信しかできないように見えますが，CLK線やDATA線はオープンコレクタでドライブされているので，キーボードやマウスなどのデバイス側とホスト側の両方が信号をドライブすることができます．またCLK線とDATA線を後述するようなルールで制御することで，デバイス→ホストやホスト→デバイスの両方向のデータ転送が可能になっています．

〔図1〕システムコントローラ(キーボード&マウスインターフェース)の構成

〔写真1〕システムコントローラ PCI ボードと M16C マイコンボード外観

(a) システムコントローラ PCI ボード

(b) M16C マイコンボード部(試作壱号)

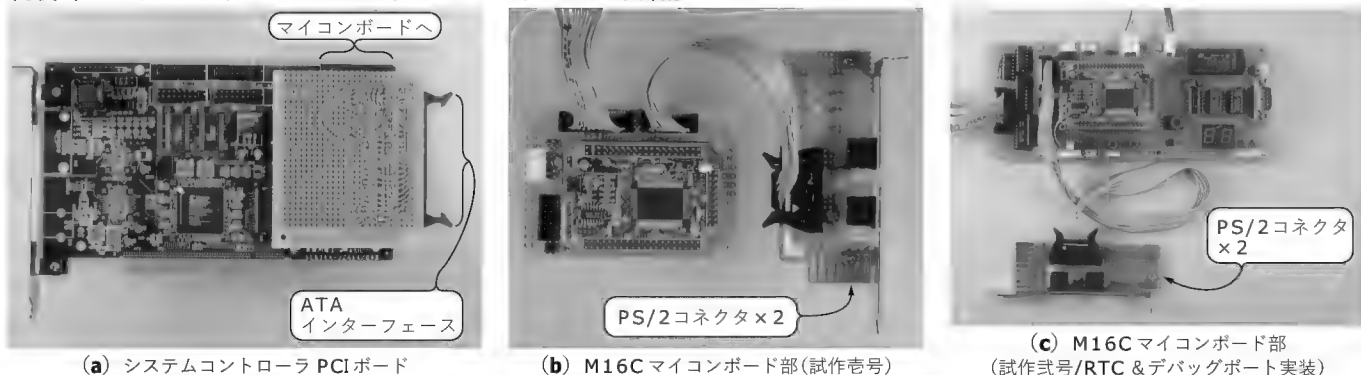
(c) M16C マイコンボード部
(試作弐号/RTC＆デバッグポート実装)

● PS/2 インターフェースのデータ通信

図3に PS/2 インターフェースのデータ通信のようすを示します．PS/2 インターフェースの通信は非常に速度が遅く，クロック線の“H”と“L”の時間は，だいたい 30μs〜50μs と幅があります．

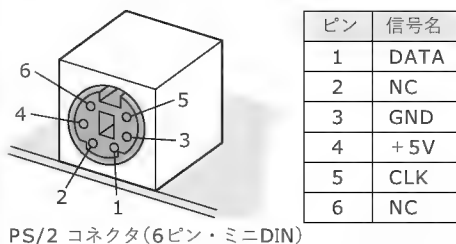
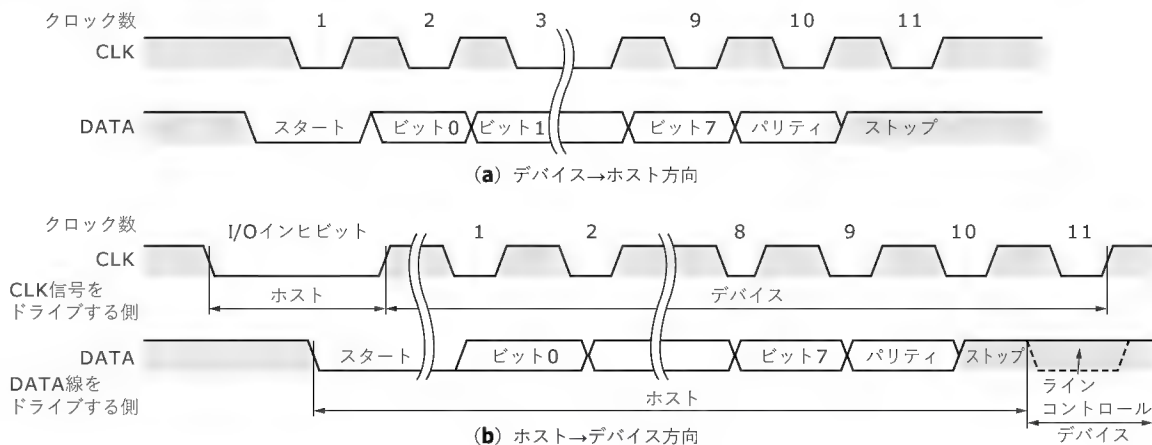
DATA 線だけを見ると，まず最初にスタートビットがあり，データが LSB から送信され，パリティビットとストップビットがあります．RS-232-C で使われる調歩同期式によく似ていますが，それぞれのビットをクロック線のタイミングで送信する部分が違います．なお，パリティは奇数パリティとなっています．

● デバイスからホスト方向の通信

デバイスからホスト方向の通信とは，ホストがキーボードやマウスからデータを受信する場合のことです．

〔図2〕PS/2 コネクタのピン配置

ピン	信号名
1	DATA
2	NC
3	GND
4	＋5V
5	CLK
6	NC

PS/2 コネクタ(6ピン・ミニDIN)

CLK 線と DATA 線がどちらも“H”の場合は通信が行われていない状態を示します．送信するデータのあるデバイスは，DATA 線を“L”にしてスタートビットを出力します．次に CLK 線を“L”にして最初のクロックを出力します．以降，CLK 線を“L”→“H”にしたら次のデータを出力していきます．

ホストに対してデータを送信中，自分(デバイス側)が CLK 線を“H”にしているはずのタイミングで実際の CLK 線の状態が“L”だった場合は，ホスト側が CLK 線を強制的に“L”にしていることを意味します．オープンコレクタのため，どちらかが“L”を出力すると片方が“H”でも“L”になるからです．これはホスト側がデバイスに対してデータの送信中止を要求していることを意味しています．この場合は DATA 線を“H”にもどし，CLK 線が“H”になるのを待って，再度始めから送信し直します．

このように，データを出力するのはもちろんデバイス側ですが，クロックを出力するのもデバイス側である点に注目してください．ホストはデバイス側の出力するクロックにしたがってデータを受信していきます．

● ホストからデバイス方向への通信

ホストからデバイス方向への通信とは，ホストがキーボードやマウスに対してコマンドやデータを送信する場合のことです．

すでに説明したように，ホストはデバイスに対して送信中止を要求することができます．ただし，CLK が 10 クロック目(パ

〔図3〕PS/2 インターフェースのデータ通信のようす

(a) デバイス→ホスト方向

(b) ホスト→デバイス方向

リティビット送信中)になっている場合には止めることができません．10ビット以前であれば，CLK線を“L”にしてデバイスに対して送信中止を要求できます．CLK線の“H”/“L”の時間は30µs～50µsなので，その数倍以上の時間だけ“L”を出力します．

これでデバイスはデータの送信を中止したはずです．そこでホストはDATA線を“L”にしてスタートビットを出力し，次にCLK線のドライブを開放します．するとCLK線はオープンコレクタなのでプルアップ抵抗により“H”になります．これでホストがデバイスに対してコマンド送信を要求していることがわかります．

以降，デバイス側はCLK線をドライブしてクロックを出力していきます．CLK線が“H”→“L”になったらホストは次のデータを出力していきます．

このように送信停止とコマンド送信要求時はホストが信号線をドライブしますが，送信が始まってからのクロックはデバイス側が出力している点に注目してください．そのためデバイス側が何か別の処理でコマンドの受信ができないような場合は，クロック出力を止めるだけでホストも待たざるを得なくなるわけ

〔表1〕キーボードコマンド/応答データ

コマンド
リセット
[FFh]（ack）
再送
[FEh]（直前のバイト）
セットキータイプ(メイク)
[FDh]（ack）[スキャンコード]…[スキャンコード]
セットキータイプ(メイク/ブレーク)
[FCh]（ack）[スキャンコード]…[スキャンコード]
セットキータイプ(タイプマティック)
[FBh]（ack）[スキャンコード]…[スキャンコード]
セットオールキー(タイプマティック/メイク/ブレーク)
[FAh]（ack）
セットオールキー(メイク)
[F9h]（ack）
セットオールキー(メイク/ブレーク)
[F8h]（ack）
セットオールキー(タイプマティック)
[F7h]（ack）
セットデフォルト
[F6h]（ack）
デフォルトディセーブル
[F5h]（ack）

コマンド
イネーブル
[F4h]（ack）
セットタイプマティックレート/ディレイ
[F3h]（ack）[Data]（ack） Data … ビット7：“0” ビット5～6：ディレイ〔表(b)参照〕 ビット4～0：レート〔表(c)参照〕
ID読み出し
[F2h]（ack）（ID下位）（ID上位）
オルタネートスキャンコード取得
[F0h]（ack）[00h]（現在のスキャンコード値）
オルタネートスキャンコード選択
[F0h]（ack）[Data]（ack） Data … スキャンコード値(1～3のみ)
エコー
[EEh]（EEh）
キーボードLED点灯制御
[EDh]（ack）[Data]（ack） Data … ビット7～4：“0” ビット3：カナLock(AXキーボードのみ) ビット2：Caps Lock ビット1：Num Lock ビット0：Scroll Lock “1”で点灯/“0”で消灯

注：[ ]はホスト→デバイスのコマンド．( )はデバイス→ホストの応答データ．
(ack)はコマンドが正常に受信されればACK(FAh)，パリティエラーなどの場合は再送(EEh)．

(a) キーボードコマンド一覧

ビット5～6	時　間
00	250ms
01	500ms
10	750ms
11	1000ms

(b) ディレイ設定値

ビット4～0	タイプマティックレート
00000	30.0
00001	26.7
00010	24.0
00011	21.8
00100	20.0
00101	18.5
00110	17.1
00111	16.0
01000	15.0
01001	13.3
01010	12.0
01011	10.9
01100	10.0
01101	9.2
01110	8.6
01111	8.0

ビット4～0	タイプマティックレート
10000	7.5
10001	6.7
10010	6.0
10011	5.5
10100	5.0
10101	4.6
10110	4.3
10111	4.0
11000	3.7
11001	3.3
11010	3.0
11011	2.7
11100	2.5
11101	2.3
11110	2.1
11111	2.0

(c) レート設定値

応答	データ
オーバラン	00h/FFh
コマンド再送	FEh
BAT(自己診断)異常終了	FCh
アクノリッジ(ACK)	FAh
エコー応答	EEh
BAT正常終了	AAh
キーボードID	83ABh/84ABh/86ABh

(d) キーボード応答データ

〔表2〕マウスコマンドと応答データ

リセット	
［FFh］（ack）	
再送	
［FEh］（ack）	
セットデフォルト	
［F6h］（ack）	
デフォルトディセーブル	
［F5h］（ack）	
イネーブル	
［F4h］（ack）	
セットサンプリングレート	
［F3h］（ack）［Data］（ack） Data … サンプリングレート	0Ah ：10サンプリング/秒 14h ：20サンプリング/秒 28h ：40サンプリング/秒 3Ch ：60サンプリング/秒 50h ：80サンプリング/秒 64h ：100サンプリング/秒 C8h ：200サンプリング/秒
ID読み出し	
［F2h］（ack）（マウスID）	
セットリモートモード	
［F0h］（ack）	
セットラップモード	
［EEh］（ack）	
リセットラップモード	
［ECh］（ack）	
リードデータ	
［EBh］（マウスデータ）	
セットストリームモード	
［EAh］（ack）	
ステータスリクエスト	
［E9h］（動作モードなど）（解像度）（サンプリングレート）	
セットレゾリューション（解像度）	
［E8h］（ack）［Data］（ack） Data … 00h ：1カウント/mm	01h ：2カウント/mm 02h ：4カウント/mm 03h ：8カウント/mm
セットスケーリング	
［E7h］（ack）	
リセットスケーリング	
［E6h］（ack）	

（a）マウスコマンド一覧

コマンド再送	FEh
BAT（自己診断）異常終了	FCh
アクノリッジ（ACK）	FAh
BAT正常終了	AAh

（b）キーボード応答データ

です．なお，**図3(b)**の最後にあるラインコントロールビットは，ストップビットに相当するクロックで，DATA線が"L"だった場合など，デバイス側からホストに対して異常発生を通知するものです．

また，CLK線とDATA線を同時に"L"にして数十ms以上の時間固定すると，ホスト側からデバイスにハードウェア的にリセットを通知することができます．

以上の通信は，キーボードでもマウスでも同様です．異なるのは，この通信でやり取りされるデータやコマンドです．

● キーボードコマンドと応答データ

表1（前頁）にキーボードコマンドと応答データを示します．まず表中の用語について説明します．メイクとはキーを押したとき，ブレークとは離したとき，タイプマティックとはオートリピートのことを意味します．スキャンコードとは，キーボードのどのキーを押したときにどんなコードが出力されるかを定義したコードのことで，PC/AT互換機で使われるキーボードでは3種類のコードセットが決められています．デフォルトで使われているのはスキャンコードセット2です．

セットタイプマティックレート/ディレイの設定コマンドとは，キーを押して最初のキー出力からオートリピートが始まるまでのディレイと，オートリピートのレート（秒間文字数）を設定するコマンドです．

セットキータイプとは，指定した各キーをメイクのみ，メイク/ブレーク，またはオートリピートありの動作に設定するコマンドで，セットオールキーとは，すべてのキーを同様にいずれかの動作に設定するコマンドです．ただし，これらのコマンドで影響を受けるのは，スキャンコードセットが3の場合のみです．

また，ID読み出しコマンドでキーボードからIDを読み出せます．IDの種類にはいくつかあるのですが，残念ながらそのIDからは，日本語キーボードと英語キーボードを識別できるわけではありません．これらのキーボードの判定は，手動で指定する以外に方法がないようです．

● キーボードデータのフォーマット

キーボードから送信されるコードは，そのとき動作しているスキャンコードセットにより異なります．デフォルトで使われているのはスキャンコードセット2ですが，詳細については後述するキーコード変換方法の節で説明します．

● マウスコマンドと応答データ

表2にマウスコマンドと応答データを示します．表中にもあるようにPS/2マウスには，リセットラップモード，セットリモートモード，セットストリームモード，セットラップモードの四つのモードがあります．

マウスの電源ON時やリセットコマンド時にはリセットラップモードに入ります．するとマウスは自己診断を行い，ホストに完了コード（AAh）とマウスID（00h）を返して，次のようなデフォルトの状態になります．

- サンプリングレート ：100サンプリング/秒
- 解像度 ：4カウント/mm
- スケーリング ：リニアスケーリング
- データ転送モード ：ストリームモード

この状態でマウスはディセーブルとなり，ホストからイネーブルコマンドが送られてくるのを待ちます．

セットリモートモードはホストからコマンド（リードデータ）を発行されないとマウスのデータを送信しないモードです．

セットストリームモードはホストからコマンドを送らなくて

も，マウスの移動を検出したり，ボタンが押された場合にデータを送るモードです．

リセットラップモードとはリセットコマンドかリセットラップモードコマンド以外は，マウスに送信されたデータをそのまま返すモードです．

また，スケーリングとは，マウスをゆっくり動かしたときは移動量を小さく，すばやく動かしたときは移動量を大きくする動作で，セットストリームモードの時のみ有効です．

● マウスデータのフォーマット

表3に移動量やボタンの状態を示すマウスデータのフォーマットを示します．現在PS/2マウスには，ごく標準的な2ボタンマウスや，ホイール付きマウス，ホイール付きの5ボタンマウスがあります．ホイール付きマウスや5ボタンマウスは，通常の手順でマウスを初期化しても，標準2ボタンマウスとしてしか動作しません．

ホイール機能や5ボタン機能を生かすには，セットサンプリングレートコマンドを次のように連続して発行します．

- セットサンプリングレート：200
- セットサンプリングレート：100
- セットサンプリングレート：80

この次にID読み出しコマンドを発行し，その値が03hの場合は，ホイール付きマウスであると判定でき，**表3(b)**のフォーマットでデータを送ってきます．ホイール付きマウスでない場合は，IDを取得しても00hのままとなります．

さらに同様に，

- セットサンプリングレート：200
- セットサンプリングレート：200
- セットサンプリングレート：80

とセットサンプルレートコマンドを発行してからIDを取得し，その値が04hであれば，ホイール付き5ボタンマウスと判定でき，**表3(c)**のフォーマットでデータを送ってきます．

〔表3〕マウスデータのフォーマット

ビット	7	6	5	4	3	2	1	0
バイト1	Y オーバフロー	X オーバフロー	Y 符号ビット	X 符号ビット	“1”	中ボタン	右ボタン	左ボタン
バイト2	X 移動量							
バイト3	Y 移動量							

(a) 標準2ボタン(3ボタン)マウス

ビット	7	6	5	4	3	2	1	0
バイト1	Y オーバフロー	X オーバフロー	Y 符号ビット	X 符号ビット	“1”	中ボタン	右ボタン	左ボタン
バイト2	X 移動量							
バイト3	Y 移動量							
バイト4	Z 移動量							

(b) ホイール付きマウス

ビット	7	6	5	4	3	2	1	0
バイト1	Y オーバフロー	X オーバフロー	Y 符号ビット	X 符号ビット	“1”	中ボタン	右ボタン	左ボタン
バイト2	X 移動量							
バイト3	Y 移動量							
バイト4	“0”	“0”	第5ボタン	第4ボタン	Z 移動量			

(c) ホイール付き5ボタンマウス

注：いずれもボタンが押されているとき“1”

2 M16Cマイコンのシリアルコントローラ仕様とPS/2インターフェース

● SIOのクロック同期モードの機能

今回採用したM16Cマイコンには3チャネルのシリアルコントローラ(以下SIO)があり，チャネル0と1はほぼ同じ機能で，チャネル2が若干仕様が異なるSIOになっています．

図3でわかるように，DATA線は調歩同期式に見えますが，データ転送時の各クロックがデバイス側の出力するクロックに同期していることから，受信時はCLKの立ち上がりでDATAをサンプリング，送信時はCLKの立ち下りで1ビットずつ出力するクロック同期式モードを使うしかないようです．

しかし，M16CのSIOはクロック同期モードで使用すると，スタートビットやパリティは付加できない仕様になっています．クロック同期モードを使いつつ，スタートビットやパリティを付加する工夫が必要です．

● ポートやカウンタも使った裏技回路？

当初はマイコンの外に74TTLを1，2個並べて，オープンドレイン出力やDATA線を送信/受信に切り分ける回路を構成していました．しかしせっかく高機能な周辺コントローラを内蔵した組み込みマイコンを使っているので，なんとか外付け回路なしでPS/2デバイスを接続できないものか，ない知恵をあれこれ絞ってみました．

ここで筆者が注目したのが，TxD0とTxD1をオープンドレインで出力できる機能です．これが使えると，TxDとRxDを直接ワイヤード接続してDATA線にすることができ，外付けにオープンコレクタのドライバを接続せずに済みます．

また，M16Cには豊富なパラレルI/Oポートやカウンタ/タイマコントローラがあります．ポートを使ってCLK線やDATA線の現在の状態を調べたり，逆に強制的に“L”に制御したり，CLK線のクロック数をカウントして現在何ビット目を送受信中かを判定できます．

M16Cの場合，オープンドレイン駆動対応のポートは一部のポートに限られていますが，“L”の出力は通常ポートでも電気的に問題ありませんし，“H”の状態を出力するにはポートの方向を入力に切り替え，プルアップ抵抗により疑似的にオープンドレイン駆動状態を実現できます．

もちろんM16Cのポートはビット単位に入出力方向を制御で

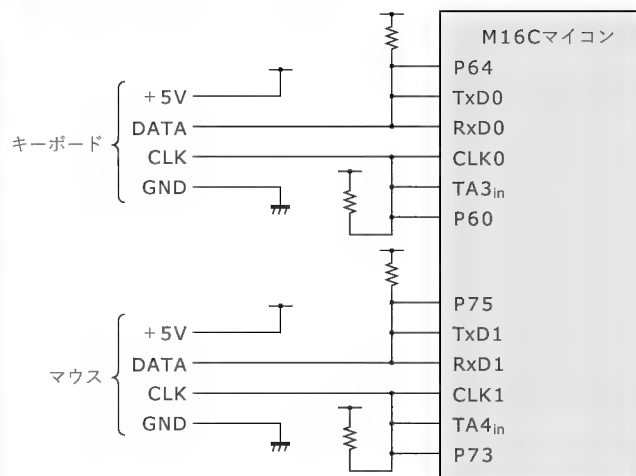
き，さらにプルアップ抵抗まで内蔵しています．ただし，マイコンに内蔵されているプルアップ抵抗は非常に抵抗値が高いので，“L”から“H”まで状態が変わるのに非常に時間がかかります．そのためクロックの立ち上がりがなだらかすぎるのか，一部のキーボードではうまく動作しない場合もあるようです．そこで，プルアップ抵抗は外付けで4.7kΩ程度のものを実装しました．

こうして考えた回路が，**図4**のPS/2インターフェース回路です．外付け部品はプルアップ抵抗のみというシンプルさです．なおM16Cのポート番号の命名は，“P”の後に8ビット単位で付けられたポート番号と，その中の何ビット目かを示す2桁(ポート10は3桁)の番号で表します．たとえば“P64”とは，ポート6のビット4を意味します．

● データ受信時の動作

図5(a)にデータ受信時の動作を示します．デバイスのハードウェアリセットは，ポートに“L”を出力することで行います．このとき，同時にタイマAチャネル3/4(以降TA3/4)のタイマを立ち上がりエッジ動作でダウンカウント，カウント値2(正確には，アンダーフローで割り込みを発生するので，2−1で1)に設定しておきます．さらにSIOは受信可能状態にしておきますが，割り込みは禁止状態に設定しておきます．ハードウェアリセットの解除はポートを入力方向にすることで，CLK線とDATA線はプルアップ抵抗により“H”の状態となります．

リセット解除によりCLK線が“L”→“H”になり1クロック，そしてデバイスから最初のデータが送られてくると，そのスタートビットのクロックを受信した時点でTA3/4のカウンタがアンダーフロー割り込みを発生させます．この割り込みでデータ受信用のシフトレジスタをクリアし，スタートビットを読み飛ばしてビット0からデータを格納させる準備をします．またRxD0/1の受信完了割り込みを有効にしておきます．さらにTA3/4のカウント値に11(同11−1で10)を設定します．次のクロックの立ち上がりが入力された時点で，ビット0のデータがシフトレジスタにサンプリングされます．

〔図4〕PS/2インターフェース回路

こうして8ビット分受信した時点でRxD0/1の受信完了割り込みが発生します．ここで受信データを取り出します．しかしデバイスは，さらにパリティとストップビットを送信してきます．RxD0/1のシフトレジスタにはビット0としてパリティ，ビット1としてストップビットが受信されてしまいますが，これは無視します．

そして次のデータのスタートビットとクロックを受信した時点で，TA3/4のカウンタがアンダーフロー割り込みを発生させます．この時点でRxD0/1のシフトレジスタのビット3にスタートビットも受信されていますが，もう一度仕切り直しということでシフトレジスタをクリアし，次のビット0から受信をはじめるわけです．

このようにRxD0/1の受信シフトレジスタにスタートビットやパリティなどのゴミ(?)が詰まってもそれをクリアし，カウンタでクロックをカウントしながら，必要な8ビットのデータ部分のみDATA線から切り出して受信するという動作をさせます．

なお，パリティを受信しても受信シフトレジスタに残ったままの状態で棄てることになるので，パリティのチェックはできません．

● データ送信時の動作

データ送信時の動作を**図5(b)**に示します．送信はもっとアクロバティックです(笑)．まずTA3/4のカウント値を取得して，現在データ受信中でないことを確認します．次にポートP60/73を出力方向にして“L”を出力します．これによりCLK線が“L”になりデバイスに対して送信中止を要求します．この時間は十分に長い時間が必要なので，タイマBチャネル0/1(以降TB0/1)をタイマ割り込みモードに設定し，次にタイマ割り込みが発生するまで出力します．また，TA3/4のカウンタ値に13(実際には13−1で12)を設定しておきます．

タイマ割り込みが発生したら，次はP64/P75を制御してDATA線も“L”にします．また，ここでTxD0/1を初期化して送信データを書き込みます．実際に送信するデータは1ビットシフトし，ビット0を“0”にしてスタートビットを付加したデータを送信バッファに書き込みます．そして，この時点ではTxD0/1のクロックの極性を立ち上がりエッジに設定しておきます．

さらに次のタイマ割り込みでP60/73を入力方向に切り替えると，プルアップ抵抗によりCLK線は“H”になります．これはクロックの立ち上がりエッジとみなされ，TxD0/1からビット0のデータが出力されます．ただし，すでにDATA線は“L”の状態なので，スタートビットの状態は変わりません．そしてTxD0/1のほかの制御レジスタはそのままで，クロックの極性設定のみ立ち下がりエッジに設定します．

あとはデバイスからクロックが出力され，クロックの立ち下がりで次々とデータビットを出力していきます．

さて，TxD0/1が8ビット分出力し終わったとしても，実際にはスタートビットを付加したデータなので，ビット6までしか送信していません．また，さらにパリティとストップビットの送信

も必要です．そこで，1バイト分のデータを送信し終えたらTxD0/1の送信（正確には送信バッファが空になった）割り込みを発生させ，ビット0に送信するデータの残りのビット7を，ビット1にあらかじめ計算しておいたパリティを，ビット2以降はすべて"1"にしたデータを送信バッファに書き込みます．

この残りのビットの出力の処理が遅くなると，ビット7以降のデータ出力が間に合わず，正しくデバイスに送信されないかもしれないと思われますが，M16CのSIOの出力バッファは，実際の出力用シフトレジスタと出力データバッファの2段構成になっていて，出力データバッファが空になった時点で割り込みが発生します．よって最初のデータが1ビットずつ出力されているときに送信割り込みが発生することになるので，あわてて残りのデータを用意することはありません．十分時間があります．

こうしてビット7とパリティ，ストップビットを送信し，最後にもう1クロックだけデバイスからクロックが出力されてデータ送信は完了します．TxD0/1の出力シフトレジスタには，ストップビット以降の"1"のデータがまだ一部出力されずに残っていますが，これは無視してかまいません．TxD0/1が"1"を出力していても，オープンドレイン駆動なので，デバイス側が"L"を出力すればDATA線は"L"になるので問題なく通信が行えます．

さて，ホストからコマンドを送信すると，必ずデバイスから応答コードが返ってきます．この応答コードのスタートビットのクロックの立ち上がりを受信した時点で，最初に設定しておいたTA3/4のアンダーフロー割り込みが発生するというわけです．以降の処理は，データ受信と同様になります．

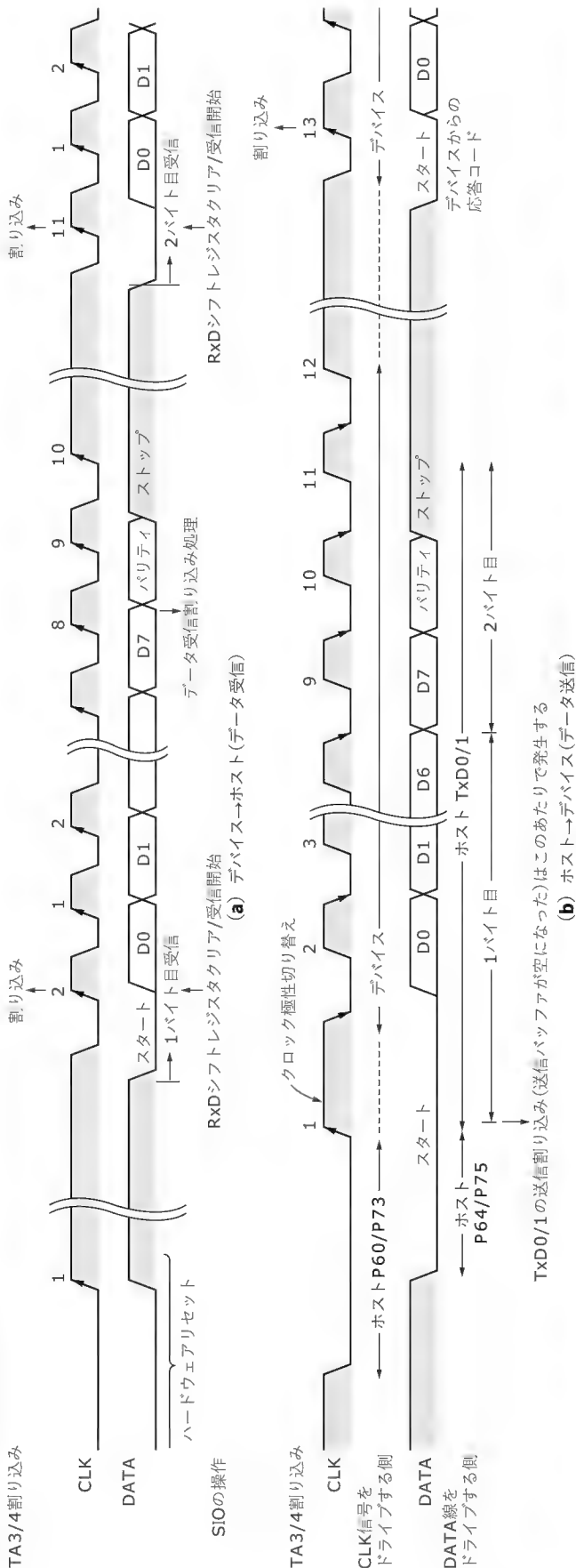
このように，データ送信の基本的アイデアは，スタートビットやパリティを付加するために2バイトにわけて送信するという点です．デバイスからデータを受信するときはホスト側の判断でパリティを無視できますが，ホストからコマンドを送るときは，正しいパリティを出力しないとデバイスがコマンドを受け取ってくれないからです．

また，スタートビットのところでクロックの極性を変更する部分は苦肉の策です．**図3**を見るとわかるように，データ送信時はCLK線を先に"L"にして送信停止要求を出すところから始まり，CLK線が"L"の途中でDATA線を"L"にしてスタートビットが開始される格好になります．データ送信時の基本である「立ち下がりエッジで出力」という設定を，送信開始時点では設定することができないわけです．

本来，すでにシリアルデータ送受信中の段階でクロック極性を変更するなど，正常な使い方とはいえません．今のバージョンのM16Cでは動作していますが，M16Cのロットが変わったら，もしかしたら動作しなくなる可能性もあります．まぁ，そのときはそのときです（笑）．

3 システムキーボード/システムマウス仕様

たとえば，キーボードコマンドと呼んだ場合，ホストCPUか

〔図5〕PS/2データ送受信の動作

〔表5〕PS/2キーボードスキャンコードとマトリクスコード対応表

キーマトリクスコード(16進数)	日本語キートップ	英語キートップ	PS/2メイクコード	キーマトリクスコード(16進数)	日本語キートップ	英語キートップ	PS/2メイクコード
01	半角/全角	~ / '	0E	56	PageDown	PageDown	E0 7A
02	! / 1	! / 1	16	57			
03	“ / 2	@ / 2	1E	58			
04	# / 3	# / 3	26	59	→	→	E0 74
05	$ / 4	$ / 4	25	5A	Num Lock	Num Lock	77
06	% / 5	% / 5	2E	5B	7	7	6C
07	& / 6	^ / 6	36	5C	4	4	6B
08	' / 7	& / 7	3D	5D	1	1	69
09	(/ 8	* / 8	3E	5E			
0A	) / 9	(/ 9	46	5F	/	/	E0 4A
0B	0	) / 0	45	60	8	8	75
0C	= / -	_ / -	4E	61	5	5	73
0D	~ / ^	+ / =	55	62	2	2	72
0E	/ ¥		6A	63	0	0	70
0F	BackSpace	BackSpace	66	64	*	*	7C
10	Tab	Tab	0D	65	9	9	7D
11	Q	Q	15	66	6	6	74
12	W	W	1D	67	3	3	7A
13	E	E	24	68	.	.	71
14	R	R	2D	69	-	-	7B
15	T	T	2C	6A	+	+	79
16	Y	Y	35	6B			
17	U	U	3C	6C	Enter(10キー)	Enter(10キー)	E0 5A
18	I	I	43	6D			
19	O	O	44	6E	ESC	ESC	76
1A	P	P	4D	6F			
1B	` / @	{ / [	54	70	F1	F1	05
1C	{ / [	} /]	5B	71	F2	F2	06
1D				72	F3	F3	04
1E	CapsLock	CapsLock	58	73	F4	F4	0C
1F	A	A	1C	74	F5	F5	03
20	S	S	1B	75	F6	F6	0B
21	D	D	23	76	F7	F7	83
22	F	F	2B	77	F8	F8	0A
23	G	G	34	78	F9	F9	01
24	H	H	33	79	F10	F10	09
25	J	J	3B	7A	F11	F11	78
26	K	K	42	7B	F12	F12	07
27	L	L	4B	7C	Print Screen	Print Screen	E0 7C 84
28	+ / ;	: / ;	4C	7D	Scroll Lock	Scroll Lock	7E
29	* / :	” / ’	52	7E	Pause	Pause	E1 14 (77) E0 7E
2A	} /]	¦ / ¥	5D	7F	(Win-L)	(Win-L)	E0 1F
2B	Enter	Enter	5A	80	(Win-R)	(Win-R)	E0 27
2C	Shift(L)	Shift(L)	12	81	(App)	(App)	E0 2F
2D		Macro	61	82			
2E	Z	Z	1A	83	無変換		67
2F	X	X	22	84	前候補		64
30	C	C	21	85	カタカナ		13
31	V	V	2A	86			
32	B	B	32	〜			
33	N	N	31	C3			
34	M	M	3A	C4	Power	Power	E0 37
35	< / ,	< / ,	41	C5	Sleep	Sleep	E0 3F
36	> / .	> / .	49	C6	Wake-up	Wake-up	E0 5E
37	? / /	? / /	4A	C7			
38	_ / ¥		51	C8	E-mail	E-mail	E0 48
39	Shift(R)	Shift(R)	59	C9	WWW Home	WWW Home	E0 3A
3A	Ctrl(L)	Ctrl(L)	14	CA	WWW Favorites	WWW Favorites	E0 18
3B				CB	WWW Search	WWW Search	E0 10
3C	Alt(L)	Alt(L)	11	CC	WWW Refresh	WWW Refresh	E0 20
3D	Space	Space	29	CD	WWW Stop	WWW Stop	E0 28
3E	Alt(R)	Alt(R)	E0 11	CE	WWW Forward	WWW Forward	E0 30
3F				CF	WWW Back	WWW Back	E0 38
40	Ctrl(R)	Ctrl(R)	E0 14	D0	Media	Media	E0 50
41				D1	Play/Pause	Play/Pause	E0 34
〜				D2	Stop	Stop	E0 3B
4A				D3	Prev Track	Prev Track	E0 15
4B	Insert	Insert	E0 70	D4	Next Track	Next Track	E0 4D
4C	Delete	Delete	E0 71	D5	Volume Up	Volume Up	E0 32
4D				D6	Volume Down	Volume Down	E0 21
4E				D7	Mute	Mute	E0 23
4F	←	←	E0 6B	D8	My Computer	My Computer	E0 40
50	Home	Home	E0 6C	D9	Calculator	Calculator	E0 2B
51	End	End	E0 69	DA	Screen Saver	Screen Saver	E0 4B
52				DB	Rec	Rec	E0 1C
53	↑	↑	E0 75	DC	Rew	Rew	E0 43
54	↓	↓	E0 72	DD	Minimize	Minimize	E0 22
55	PageUp	PageUp	E0 7D	DE	Eject	Eject	E0 1D

らマイコンへのキーボード制御コマンドなのか，PS/2キーボードへマイコンが出力するキーボードコマンドか不明瞭になるので，以降では，マイコンとPS/2キーボード側の生のコマンドやデータを示す場合はPS/2キーボード○○，マイコンとホストCPUの間のコマンドやデータを示す場合はシステムキーボード○○と呼ぶことにします．

● スキャンコードとコード変換

PS/2キーボードにはいくつかスキャンコードセットがあり，デフォルトではスキャンコードセット2が使われていることは説明しました．スキャンコードセット2は，基本的にはキーを押したときに1バイトのメイクコードが，キーを離したときにブレークコードプリフィックスとしてF0hが付加されたコードが送られます．たとえば“A”キーを押して離すと，押した瞬間に1Chが，離した瞬間にF0h，1Chのデータがホストに出力されます．またオートリピートが働いたときは，メイクコードのみが連続して出力されます．一部特殊なキーは，メイクコードも複数バイトからなり，可変長のデータフォーマットになっています．これをマイコンを使って，すべてのキーとメイク/ブレークで統一的なコード(キーマトリクスコードと呼ぶことにする)に変換します．

また，せっかく間にマイコンを入れて変換しているので，キーマトリクスコードに変換するだけでなく，直接ASCIIコードに変換するモードも設け，より低レベルのI/O処理でも直接ASCIIコードが使えるようにも考えてみました．

それともう一つ，PS/2キーボードからのデータを直接ホストCPUで受信したり，PS/2キーボードコマンドを直接送信するためのダイレクトモードも実装することにしました．ダイレクトモードではマイコンはコマンドやスキャンコードの変換なしに，PS/2キーボードとホストCPUの間のデータを受け渡しします．

● キーマトリクスコード

PS/2キーボードのキーの個数は，106/109日本語キーボードなどと呼ばれるように，通常は128個もありません．メイク/ブレークを示すビットを最上位ビットに割り当てて8ビットで収めることも可能です．しかし，将来的な拡張を見越して，キーマトリクスコードは8ビット分を確保し，メイク/ブレークや，

その他のフラグ情報として上位8ビットを追加した，16ビット固定長を採用することにしました．

表5にPS/2キーボードのメイクコード(スキャンコードセット2のとき)とキートップの文字，そして変換後のキーマトリクスコードの一覧を示します．たとえば“A”キーを押した場合，PS/2キーボードからの1Chをホスト CPUへは001Fhとして，離した場合はPS/2キーボードからのF0h 1ChをホストCPUへは801Fhと変換します．またオートリピート時は001Fhを連続して出力します．

表の形では見にくいので，**図6**に109日本語キーボードにおけるキーマトリクスコードの配列のようすを示します．

● システムキーボードコマンド

表1に示したように，PS/2キーボードを制御するにはさまざまなPS/2キーボードコマンドがありました．その中にキーボードID読み出しコマンドがあります．しかし，このコマンドでは101英語キーボードと106日本語キーボードの判定はできません．よって，キーボードの種類はホストから設定しなければなりません．

また，人によってはオートリピートまでの時間(ディレイ)を短くしたり，リピートを早く(レートを多く)したい人もいるでしょう．これらの設定は，ホストCPUからも設定できるように，システムキーボードコマンドとして実装します．

キーボードのLEDは，通常はマイコンが自動的に点灯制御をするようにしましたが，ホスト側に本格的なOSを載せた場合など，キーボードを密に制御する本格的なドライバを載せたときに，ホストからLEDの点灯状態を制御できるようにも考えました．

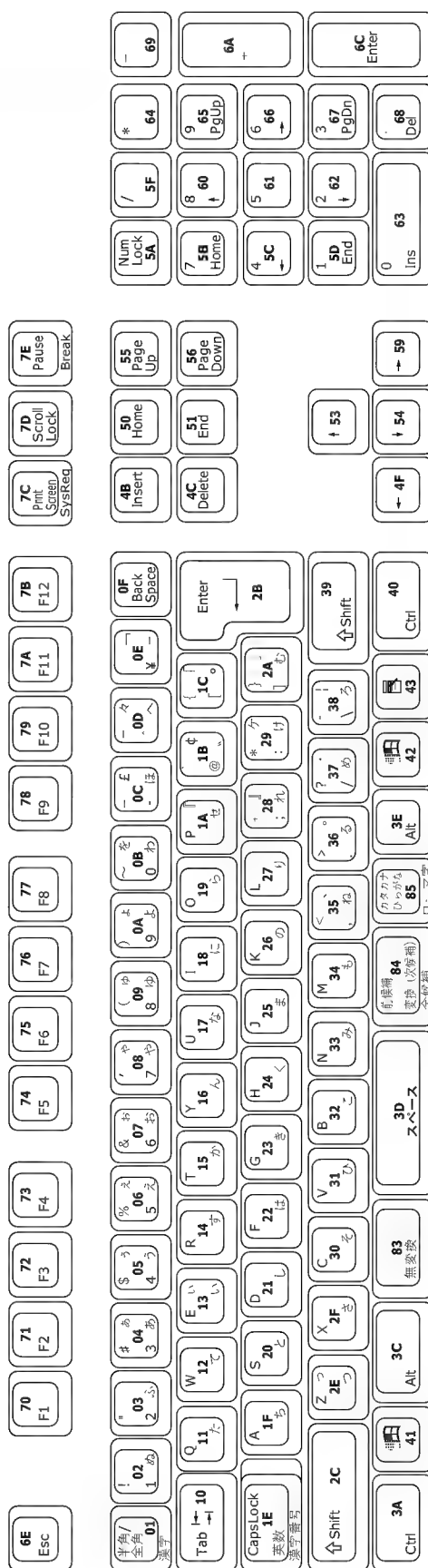
● システムマウスデータのフォーマット

マウスから送られてくるデータは，標準マウスの場合は3バイト，ホイール付きマウスの場合は4バイトであることは説明しました．これもデータが可変長になっているので，そのままではホスト側で統一的な制御ができなくなります．

そこでマウスについても，マイコンでマウスの種類の自動判定を行い，受信したデータのフォーマットを変換して，標準マウスやホイール付きマウスでも同一のデータフォーマットでホストに返すようにします．

システムマウスデータのフォーマットは，まったく新規に規定してもよいのですが，それぞれのマウスのデータフォーマットの先頭3バイトは互換性があることもあり，**表3(c)**のホイール付き5ボタンマウスのフォーマットを基本に一部変更を加えたフォーマットとしました．標準マウスの場合は，Z軸方向の移動量は常にゼロで，第4ボタンと第5ボタンは押されていない状態に変換します．また，ホイール付きマウスの場合は，Z軸方向の移動量を4ビットで表現できる範囲にクリッピングし，第4ボタンと第5ボタンは押されていない状態に変換することで，同じくフォーマットを合わせます．

● システムマウスコマンド

PS/2マウスも基本的にはマイコンにより自動的に初期化しま

〔図6〕109日本語キーボードのマトリクスコード一覧

すが，サンプリングレートや解像度，スケールの設定は，ホストからも任意に指定できるようにコマンドを設けます．また，自動判定したマウスの種類をホスト側でも取得できるようなシステムマウスコマンドも用意しました．

また，マイコンによるPS/2マウスの自動制御モードのほかに，ホスト側が直接マウスのPS/2インターフェースに対してコマンドを送信したり，受信したデータをダイレクトに取得できるよう，ダイレクトモードも用意しました．

4 マイコンファームウェアの処理概要

● 割り込み駆動はPS/2デバイスのみ

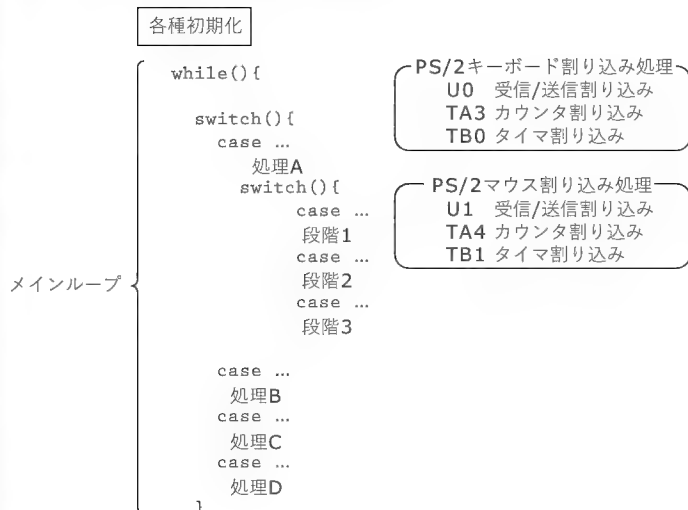
じつは当初，PS/2デバイスとの通信部分は1ビット単位でソフトウェアでポーリングする方法を採っていました．しかしあまりに負荷が重く，今後電源制御など他の処理が増えたとき，ビットの取りこぼしなどが発生する可能性も考えられます．そこで最終的にPS/2デバイスとのデータ送受信は，すでに説明したようなシリアルコントローラ/カウンタ/タイマ/ポート機能を総動員し，割り込みで対処する方法を採りました．

しかし，キーボードとマウスは同時に動かすので，同時に割り込みが発生することも十分に考えられます．

● ホストCPUからのコマンド処理はポーリング

そこでホストCPUからのコマンド処理は，ポーリングで処理する方針とします．つまり，PS/2デバイスとの通信が忙しいときは，割り込みが多発してメインループの実行が遅くなってもかまわないとするのです．

図7に示すように，メインは大きなループ構成とし，それぞれの処理はできるだけ小さな処理単位として，できるだけメインループを速く回すように心がけます．

とはいえ，ホストCPUがコマンドを発行してからそれが実行されるまで，かなりの時間が必要になります．そこで，ホストからのほとんどのコマンドは発行後，ビジィ/レディフラグを見る，もしくは割り込みでコマンドの実行終了を判定するプロトコルを採用することで，マイコンがそのコマンドの処理にとりかかれなくても，ホスト側を待たせることができるようにします．そのためのしくみは後述するPCIデバイスの中にハード的に用意します．

〔図7〕マイコンファームウェアの基本構造

```
各種初期化

メインループ
  while(){
    switch(){
      case ...
        処理A
        switch(){
          case ...
            段階1
          case ...
            段階2
          case ...
            段階3
      case ...
        処理B
      case ...
        処理C
      case ...
        処理D
  }

PS/2キーボード割り込み処理
  U0   受信/送信割り込み
  TA3  カウンタ割り込み
  TB0  タイマ割り込み

PS/2マウス割り込み処理
  U1   受信/送信割り込み
  TA4  カウンタ割り込み
  TB1  タイマ割り込み
```

5 システムコントローラPCIデバイスのハードウェア

● PCIバスとマイコン(M16C)の橋渡し

M16Cの外部バスではPCIバスと直接接続することはできないので，PCIバスとM16Cの間を橋渡しするPCIデバイスが必要です．ただし，M16Cの外部バスアクセスをPCIバスアクセスにブリッジさせるなどの処理を行う必要はありません．PCI側から見えるレジスタとマイコン側から見えるレジスタというように，PCIデバイス内部に2ポートレジスタを用意してアクセスさせるだけです(**リスト1**，**図8**)．とはいえ，単なる2ポートレジスタというわけではありません．

たとえば，ホストからコマンドを発行したときのコマンド実行中を示すビジィビットの制御を考えます．単純に考えた場合，コマンドを受け取るマイコン側がビジィビットを立てて，コマンドの実行が終わったらクリアするという処理が考えられます．しかし，マイコン側がすぐにはコマンドを受け取れない場合，ホスト側はコマンドを発行した後すぐにビジィビットを読み出しても，ビジィビットがクリア状態なのでコマンドの実行が終了したものと誤認識してしまう可能性もあります．

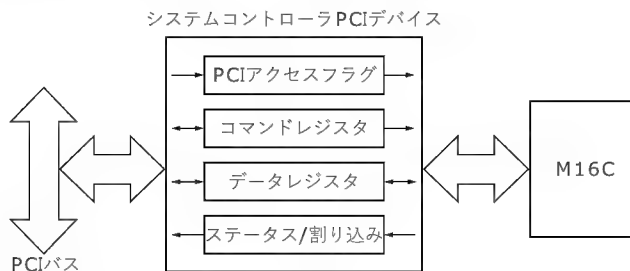
とくに今回は，ファームウェアの構造で説明したように，PCI側からのアクセスをポーリングで処理するようにしたので，PCI側からコマンドを発行してからマイコンがそれを受け取るまでのタイムラグが十分考えられます．

そこで，コマンドをコマンドレジスタに書き込むと同時に，コマンドビジィビットがセットされるようなハードウェアを構成します．これならば，マイコンが忙しくてコマンドをすぐに受け取れない状態でも，コマンド発行と同時にビジィ状態が示されるので，ホスト側がコマンドの実行終了を正しく認識することができます．

● 割り込み制御回路

また，今回はビジィやレディビットをポーリングで調べるこ

〔図8〕システムコントローラPCIデバイスの構造

システムコントローラPCIデバイス
PCIアクセスフラグ
コマンドレジスタ
データレジスタ
ステータス/割り込み
PCIバス
M16C

とも可能ですが，ホスト側へはこれらの状態の変化を，すべて割り込みでも通知できるようなハードウェアにします．ビジィやレディと連動して，割り込み出力が許可状態であれば，ホストに対して割り込みを出力する(INT*n#* をアサートする)回路も実装します．

さらにPCIデバイスなので，割り込みは共有可能に設計しなければなりません．他の割り込みと共有されると，自分が割り込みを出力していないのに，割り込み処理ルーチンが呼び出される場合があるので，必ずステータスレジスタを用意して割り込みを出力しているか否かを判定できるようにします．割り込みステータスが立っていないのに割り込み処理ルーチンが呼ばれた場合は，割り込みを共有しているほかのデバイスが割り込みを出力していると判断して，自分のデバイスに対しての割り込み処理はとくに行わずに，そのシステムで規定された割り込み処理の終了処理(たいていの場合はそのままリターン)をします．

〔リスト1〕システムコントローラのVHDLソース

```
                              ～中略～
-- ********** LOCAL IO ACCESSA時の動作 ********** --
when LOCAL IO ACCESS =>           --

if (PCI BusCommand(0) = '1' ) then    -- I/Oライトサイクル

    case PCI Address(5 downto 2) is
         when "0000" => -- +00h 割り込み制御レジスタ
              if (C nBE(1) = '0') then
                   INT Mask <= PCIAD(15 downto  8);
              end if;

         when "0010" => -- +08h キーボード制御レジスタ/マウス制御レジスタ
              if (C nBE(1) = '0') then
                   KEY Command(15 downto  8)
                                        <= PCIAD(15 downto  8);
              end if;
              if (C nBE(0) = '0') then
                   KEY Command( 7 downto  0)
                                        <= PCIAD( 7 downto  0);
              end if;
              if (C nBE(1) = '0') or (C nBE(0) = '0') then
                   KEY WriteFlg <= '1';
                             -- キーボード制御レジスタ書き込みフラグセット
              end if;

         when "0011" => -- +0Ch マウス制御レジスタ
              if (C nBE(1) = '0') then
                   MOUSE Command(15 downto  8)
                                        <= PCIAD(15 downto  8);
              end if;
              if (C nBE(0) = '0') then
                   MOUSE Command( 7 downto  0)
                                        <= PCIAD( 7 downto  0);
              end if;
              if (C nBE(1) = '0') or (C nBE(0) = '0') then
                   MOUSE WriteFlg <= '1';
                                -- マウス制御レジスタ書き込みフラグセット
              end if;

                              ～中略～

         when others =>
              null;
    end case;

         else                                       -- I/Oリードサイクル

    case PCI Address(5 downto 2) is
         when "0000" => -- +00h 割り込み制御レジスタ
              PCIAD Port(31 downto 16) <= (others => '0');
                                                          -- +02h~03h
              PCIAD Port(15 downto  8) <= INT Mask;  -- +01h
              PCIAD Port( 7 downto  0) <= INT Stat;  -- +00h

         when "0010" =>
              -- +08h キーボード制御レジスタ/マウス制御レジスタ
              PCIAD Port(31) <= MOUSE DataEn;    -- +0Bh
              PCIAD Port(30) <= MOUSE CmdBusy;
              PCIAD Port(29 downto 24) <= MOUSE Signal;
              PCIAD Port(23) <= KEY DataEn;      -- +0Ah
              PCIAD Port(22) <= KEY CmdBusy;
              PCIAD Port(21 downto 16) <= KEY Signal;
              PCIAD Port(15 downto  0) <= KEY Data;  -- +08h~09h
              if (C nBE(0) = '0') or (C nBE(1) = '0') then
                   KEY DataEn CLR <= '1';
                                  -- キーボードデータイネーブルクリア信号
              end if;

         when "0011" => -- +0Ch マウス制御レジスタ
              PCIAD Port(31 downto  0) <= MOUSE Data; -- +0Ch~0Fh
              MOUSE DataEn CLR <= '1';
                                    -- マウスデータイネーブルクリア信号
                              ～中略～

         when others =>
              PCIAD Port(31 downto  0) <= (others => '0');

    end case;

end if;

LOCAL DTACK <= '1' ;
                         -- ローカルバスシーケンサ データ転送完了フラグ セット
LOCAL NEXT STATE := LOCAL STATE COMP;
                              ～中略～
```

(a) PCI側レジスタ

```
                              ～中略～
SYSCTRL Read equ : process ( int KWRn, KRDn, KCSn, KA )
begin
    if (KRDn = '0') and (int KWRn = '1') and (KCSn = '0') then
         case KA(5 downto 0) is

              when "000000" =>    -- +00h PCIバスリセットフラグレジスタ
                   KD Port(7) <= PCIBus RST;
                   KD Port(6 downto 0) <= M16C ReadCount;

              when "000001" =>    -- +01h PCIアクセスフラグレジスタ
                   KD Port(7) <= SYSTEM WriteMark;
                   KD Port(6) <= LED WriteMark;
                   KD Port(5) <= POWER WriteMark;
                   KD Port(4) <= TK AccStartMark;
                   KD Port(3) <= MOUSE WriteMark;
                   KD Port(2) <= KEY WriteMark;
                   KD Port(1 downto 0) <= (others => '0');

              when "001000" =>
                         -- +08h キーボード制御レジスタ(コマンド下位)
                   KD Port <= KEY Command( 7 downto  0);
              when "001001" =>
                         -- +09h キーボード制御レジスタ(コマンド上位)
                   KD Port <= KEY Command(15 downto  8);
              when "001010" =>    -- +0Ah キーボード制御レジスタ
                   KD Port(7) <= KEY DataEn;
                   KD Port(6) <= KEY CmdBusy;
                   KD Port(5 downto 0) <= KEY Signal;

              when "001011" =>    -- +0Bh マウス制御レジスタ
                   KD Port(7) <= MOUSE DataEn;
                   KD Port(6) <= MOUSE CmdBusy;
                   KD Port(5 downto 0) <= MOUSE Signal;
              when "001100" =>
                         -- +0Ch マウス制御レジスタ(コマンド下位)
```

(b) M16側レジスタ

〔リスト1〕システムコントローラのVHDLソース(つづき)

```
                KD Port <= MOUSE Command( 7 downto  0);
            when "001101" =>
                                    -- +0Dh マウス制御レジスタ(コマンド上位)
                KD Port <= MOUSE Command(15 downto  8);

                            ～中略～

            when others => -- その他のレジスタ
                KD Port <= (others => '0');  -- すべて0を返す

        end case;
    end if;
end process SYSCTRL Read equ;

SYSCTRL Write equ : process ( int KWRn, KRDn, KCSn, KA )
begin
    if (int KWRn'event and int KWRn = '1') then
        if (KRDn = '1') and (KCSn = '0') then

            case KA(5 downto 0) is
                when "000000" =>
                                    -- +00h M16C→PCIレジスタ リセット処理
                    KEY Signal <= "000011";
                                            -- モード3(デフォルト)
                    KEY Data <= (others => '0');
                    MOUSE Signal <= "000001";
                                            -- モード1(デフォルト)
                    MOUSE Data <= (others => '0');

                            ～中略～

                when "001000" =>
                                -- +08h キーボード制御レジスタ(データ下位バイト)
                    KEY Data( 7 downto  0) <= KD;
                when "001001" =>
                                -- +09h キーボード制御レジスタ(データ上位バイト)
                    KEY Data(15 downto  8) <= KD;
                when "001010" =>    -- +0Ah キーボード制御レジスタ
                    KEY Signal <= KD(5 downto 0);

                when "001011" =>    -- +0Bh マウス制御レジスタ
                    MOUSE Signal <= KD(5 downto 0);
                when "001100" =>
                                -- +0Ch マウス制御レジスタ(データバイト0)
                    MOUSE Data( 7 downto  0) <= KD;
                when "001101" =>
                                -- +0Dh マウス制御レジスタ(データバイト1)
                    MOUSE Data(15 downto  8) <= KD;
                when "001110" =>
                                -- +0Eh マウス制御レジスタ(データバイト2)
                    MOUSE Data(23 downto 16) <= KD;
                when "001111" =>
                                -- +0Fh マウス制御レジスタ(データバイト3)
                    MOUSE Data(31 downto 24) <= KD;

                            ～中略～

                when others => -- その他のレジスタ
                    null;

            end case ;
        end if ;
    end if ;
end process SYSCTRL Write equ;
```

(b) M16側レジスタ(つづき)

〔表6〕システムコントローラ(キーボード/マウス関連)レジスタ一覧

オフセット	名　称
+00h	割り込みステータスレジスタ
+01h	割り込みマスクレジスタ
+02h～+07h	(予約)
+08h～+09h	システムキーボードデータ/コマンドレジスタ
+0Ah	システムキーボード制御レジスタ
+0Bh	マウス制御レジスタ
+0Ch～0Fh	マウスデータ/コマンドレジスタ
+10h～13h	TKアドレス/制御レジスタ
+14h～17h	TKデータレジスタ
+18h～1Bh	電源制御レジスタ
+1Ch～1Fh	LED点灯制御レジスタ
+20h～3Bh	(予約)
+3Ch～3Dh	システムコントローラ通信制御レジスタ
+3Eh～3Fh	制御権管理レジスタ

(a) PCI側レジスタマップの概要

オフセット	名　称
+00h	PCIバスリセットフラグレジスタ
+01h	PCIバスアクセスフラグレジスタ
+02h～+07h	(予約)
+08h～+09h	システムキーボードデータ/コマンドレジスタ
+0Ah	システムキーボード制御レジスタ
+0Bh	マウス制御レジスタ
+0Ch～+0Fh	マウスデータ/コマンドレジスタ
+10h～13h	TKアドレス/制御レジスタ
+14h～17h	TKデータレジスタ
+18h～1Bh	電源制御レジスタ
+1Ch～1Fh	LED点灯制御レジスタ
+20h～3Bh	(予約)
+3Ch～3Dh	システムコントローラ通信制御レジスタ
+3Eh～3Fh	制御権管理レジスタ

(b) マイコン側レジスタマップの概要

オフセット+00h/01h以外，PCIバス側と同じ構成となっている(データレジスタの読み書き方向などは逆)

▶割り込み制御レジスタ

オフセット	サイズ(ビット)	R/W	名　称	説　明
+00h	8	R	割り込みステータスレジスタ	ビット7(拡張用) ビット6(拡張用) ビット5 電源制御割り込み ビット4 TKアクセス完了割り込み ビット3 マウス割り込み ビット2 キーボード割り込み ビット1～0(拡張用) "1"：割り込み要求中/"0"：割り込み非要求中 割り込み要求の解除は，各割り込み要因の手順に従う
+01h	8	R/W	割り込みマスクレジスタ	ビットの割り当ては割り込みステータスレジスタと同じ "1"：割り込み許可/"0"：割り込み禁止

(c) PCI側割り込み/キーボード/マウスレジスタの詳細

〔表6〕**システムコントローラ**(キーボード/マウス関連)**レジスター覧**(つづき)

▶システムキーボード制御レジスタ

オフセット	サイズ(ビット)	R/W	名称	説明
+08h	8/16 8/16	R W	システムキーボードデータレジスタ システムキーボードコマンドレジスタ	キーボード→ホストへの読み出し専用レジスタ ホスト→キーボードへの書き込み専用レジスタ データレジスタとコマンドレジスタは同じアドレスだが独立したレジスタとして実装されている
+0Ah	8	R	システムキーボード制御レジスタ	ビット7 キーボードデータレジスタ有効ビット "1"でキーボードデータレジスタに有効なデータがある キーボードデータレジスタを読み出すとクリアされる ビット6 キーボードコマンド実行ビット "0"でキーボードコマンド書き込み可能, "1"でキーボードコマンド実行中 キーボードコマンドレジスタに書き込むとセットされる ビット1～0 キーボード制御モードステータス(モード0～3)

▶システムキーボード受信データ

システムキーボードコマンド応答データ時	
ビット15	"1":キーボードコマンド応答データフラグ
ビット7～0	コマンド応答コード

▶システムキーボード受信データ

キーボード受信データ時	
ビット15	"0":キーボード受信データフラグ
ビット14	"0":メイク/"1":ブレークフラグ(オートモード0/1) 常時"0"(ASCIIモード)
ビット7～0	マトリクスコード(オートモード0/1) ASCIIコード(ASCIIモード)

▶システムキーボードコマンド/応答コード

コマンド	コマンドコード	応答コード
キーボードリセット	8000h	正常時:00h/異常時:FFh
キーボード制御モード設定	81xxh 00h モード0(ダイレクトモード) 01h モード1(ASCIIモード) 02h モード2(LED自動点灯制御なし) 03h モード3(LED自動点灯制御あり)	00h
キーボードタイプ設定	82xxh 00h 101(104)英語キーボード 10h 106(109)日本語キーボード 20h AXキーボード	00h
キーボードタイプ取得	8300h	00h 101(104)キーボード 10h 106(109)キーボード 20h AXキーボード
キーボードディレイ/レート設定	84xxh xxh ディレイ/レート設定値	正常時:00h/異常時:FFh
キーボードLED点灯制御	850xh ビット0 Scroll Lock("1":点灯/"0":消灯) ビット1 Num Lock(同上) ビット2 Caps Lock(同上) ビット3 Kana(同上)(AXキーボードのみ) 注:モード1とモード2時, LEDの点灯制御は自動で行われる(マイコンが行う)	正常時:00h/異常時:FFh

▶システムマウス制御レジスタ

オフセット	サイズ(ビット)	R/W	名称	説明
+0Bh	8	R	システムマウス制御レジスタ	ビット7 マウスデータレジスタ有効ビット "1"でマウスデータレジスタに有効なデータがある マウスデータレジスタを読み出すとクリアされる ビット6 マウスコマンド実行ビット "0"でマウスコマンド書き込み可能, "1"でマウスコマンド実行中 マウスコマンドレジスタに書き込むとセットされる ビット5～0 マウス制御モードステータス(モード0/1)
+0Ch	8/16/32	R	システムマウスデータレジスタ	マウス→ホストへの読み出し専用レジスタ
	16	W	システムマウスコマンドレジスタ	ホスト→マウスへの書き込み専用レジスタ データレジスタとコマンドレジスタは同じアドレスだが独立したレジスタとして実装されている

(c) PCI側割り込み/キーボード/マウスレジスタの詳細(つづき)

〔表6〕**システムコントローラ**(キーボード/マウス関連)**レジスター覧**(つづき)

▶システムマウスコマンド/応答コード

コマンド	コマンドコード	応答コード
マウスリセット	8000h	正常時：00h/異常時：FFh
マウス制御モード設定	81xxh 00h　モード0（ダイレクトモード） 01h　モード1	00h
マウスタイプ取得	8200h	00h　ノーマルマウス 10h　ホイール付きマウス 11h　ホイール付き5ボタンマウス
サンプルレート設定	83xxh サンプルレート 10（0Ah） 20（14h） 40（28h） 60（3Ch） 80（50h） 100（64h） 200（C8h）	正常時：00h/異常時：FFh
スケール設定	85xxh 01h　1：1 02h　2：1	正常時：00h/異常時：FFh
解像度	84xxh 00：1カウント/mm 01：2カウント/mm 02：4カウント/mm 03：8カウント/mm	正常時：00h/異常時：FFh

▶システムマウスデータレジスタ

システムマウスコマンド応答データ時	
ビット31	“1”：マウスコマンド応答データフラグ
ビット30～24	常時“0”
ビット23～8	不定
ビット7～0	コマンド応答コード

システムマウス受信データ時	
ビット31	“0”：マウス受信データフラグ
ビット30	常時“0”
ビット29	第5ボタン(“1”で押されている)
ビット28	第4ボタン(〃)
ビット27～24	Z方向移動量(−8～+7)
ビット23～16	Y方向移動量
ビット15～8	X方向移動量
ビット7	Y方向オーバフロービット
ビット6	X方向オーバフロービット
ビット5	Y方向サインビット
ビット4	X方向サインビット
ビット3	常時“1”
ビット2	中央ボタン(“1”で押されている)
ビット1	右ボタン　(　〃　)
ビット0	左ボタン　(　〃　)

(c)　PCI側割り込み/キーボード/マウスレジスタの詳細(つづき)

なお，今回の設計では，PCI側からマイコン側に対して割り込みを要求することはしていません．すでに説明したように，マイコン側はメインループでPCI側からのアクセスをポーリングで取得する構造にしているためです．

● 非同期回路設計

PCI側はPCIクロックに同期した同期設計が可能ですが，マイコン側はPCIクロックとはまったく同期していないクロックで動作しているので，非同期設計が重要になります．

● PCI側/マイコン側レジスター覧

表6(pp.110-112)にシステムコントローラのキーボード/マウス関連のレジスター覧を示します．PCI側から見た場合とマイコン側から見た場合で，基本的には同じオフセットのところに同じレジスタを配置しています．データレジスタの類は，PCI側から書き込んだコマンドをマイコン側から読み出し，マイコン側から書き込んだデータをPCI側から読み出すというように，読み書き方向が逆になります．ビジィやステータスを示す制御レジスタは，可能なかぎり同じ配置になるようにしました．

また，マイコンのコマンド実行終了は割り込みでも判定できるので，PCI側には割り込み制御レジスタを用意しています．逆にマイコン側は，ポーリングでホストから指示を判定するので，PCIアクセスフラグと名づけたステータス信号のレジスタを用意しています．

やまたけ・いちろう　来栖川電工有限会社
ふじがおか・まさのぶ　ファームウェアプログラマ

第6章

もっとも基本的なPIO転送に対応したATAインターフェース

ATAインターフェースの設計/製作

山武一朗

この章では，PIO転送対応のATAインターフェースをPCIバス上に設計/製作する．まずATAの仕様について簡単に解説し，PCIバス上にどのようにATAホストコントローラを実現するかを考察する．そしてFPGAを搭載したPCI評価用ボードを使い，もっとも基本的な転送モードであるPIO転送に対応したATAホストコントローラをHDLを使って設計し，制御ソフトウェアを作成する．（編集部）

はじめに

● ストレージも必要でしょう！

キーボード/マウス入力と画面表示出力とくれば，次にコンピュータシステムを構成する要素として重要なのはストレージでしょう．代表的なストレージインターフェースとしてIDEとSCSIが挙げられますが，価格や品種の多さや入手性を考慮すると，現状ではIDE，すなわちATAインターフェースがもっとも普及しているといえます．

そこで，ここでは，システムバスとして採用したPCIバス上に，ATAインターフェースを実装して，HDDやCD-ROMドライブを接続してみます．

1 ATAの仕様とATAホストコントローラ

ATAインターフェースを実装するとは，HDDやCD-ROMドライブに対して，アドレスやリード/ライトクロックなどの制御信号を出力する回路を設計することを意味します．ATAとはどんなバスで，ATAホストコントローラを設計する場合，どんな機能を実装しなければならないのでしょうか？

● ATA仕様とデータ転送の実体

ATAではデータ転送方式によってデータ転送を行う実体が異なり，それぞれ転送モードに名前が付いています．CPUがレジスタを読み書きしてデータ転送を行うPIO転送モード，システムに実装されているDMAコントローラを使ってデータ転送を行うシングル/マルチワードDMA転送モード，そしてPCIバスのバスマスタDMAコントローラを使ってデータを転送するUltra DMA転送モードの三つです．

PCIバス上にATAインターフェースを実装する場合，Ultra DMA転送モードがもっとも高速なデータ転送を実現できますが，技術的にも難しいものがあります．

今回はもっとも基本的な転送モードであるPIO転送モードに焦点をあてて，ATAホストインターフェースを実現してみます．

● ATAの信号

ATAの信号には，Ultra DMA転送時とそれ以外の転送時では，信号名や役割が異なるものがあります．今回はPIO転送に対応したホストなので，PIO転送時のそれぞれの信号の意味について説明します．**表1**にATAの信号名とピン配置を示します．

▶ CS[1：0]#

ATAの各レジスタにアクセスするために使用するチップセレクト信号です．DMACK#がアサートされたとき，この信号はネゲートしている必要があります．

▶ DA[2：0]

データまたはデータポートへアクセスするためのアドレス信号

〔表1〕ATAの信号名とピン配置

信号名	コネクタピン番号	ケーブル芯番号	ケーブル芯番号	コネクタピン番号	信号名
RESET#	1	1	2	2	GND
DD7	3	3	4	4	DD8
DD6	5	5	6	6	DD9
DD5	7	7	8	8	DD10
DD4	9	9	10	10	DD11
DD3	11	11	12	12	DD12
DD2	13	13	14	14	DD13
DD1	15	15	16	16	DD14
DD0	17	17	18	18	DD15
GND	19	19	20	20	(KEYPIN)
DMARQ	21	21	22	22	GND
DIOW#	23	23	24	24	GND
DIOR#	25	25	26	26	GND
IORDY	27	27	28	28	CSEL
DMACK#	29	29	30	30	GND
INTRQ	31	31	32	32	RESERVED
DA1	33	33	34	34	PDIAG#
DA0	35	35	36	36	DA2
CS0#	37	37	38	38	CS1#
DASP#	39	39	40	40	GND

〔表2〕
ATAレジスタ一覧

CS1#	CS0#	DA2	DA1	DA0	レジスタ名	
					リード	ライト
H	H	X	X	X	レジスタが選択されていない状態	
L	L	X	X	X	禁止	
L	H	L	L	L	廃止	
L	H	L	L	H	廃止	
L	H	L	H	L	廃止	
L	H	L	H	H	廃止	
L	H	H	L	L	廃止	
L	H	H	L	H	廃止	
L	H	H	H	L	Alternate Statusレジスタ	Device Controlレジスタ
L	H	H	H	H	廃止	
H	L	L	L	L	Dataレジスタ(16ビット)	
H	L	L	L	H	Errorレジスタ	Featuresレジスタ
H	L	L	H	L	Interrupt Reasonレジスタ(ATAPI)/Sector Countレジスタ(ATA)(※ATAPIのときは読み出し専用レジスタとなる)	
H	L	L	H	H	Sector Numberレジスタ(ATA)(※ATAPIのときは未使用)	
H	L	H	L	L	Byte Count LSBレジスタ(ATAPI)/Cylinder Lowレジスタ(ATA)	
H	L	H	L	H	Byte Count MSBレジスタ(ATAPI)/Cylinder Highレジスタ(ATA)	
H	L	H	H	L	Device/Headレジスタ	
H	L	H	H	H	Statusレジスタ	Commandレジスタ

です.

▶ DASP#

デバイスアクティブインジケータ(いわゆるアクセスランプ),およびデバイス1プレゼント信号がマルチプレクスされた信号です.

▶ DD[15:0]

8または16ビットのデータバスです.後述するATAレジスタのうちDataレジスタ以外は8ビットレジスタで,これらのレジスタへの読み書き時は下位の8ビットが使用されます.

▶ DIOR#

ATAレジスタを読み出すために使用するリード信号として動作します.

▶ DIOW#

ATAレジスタに書き込むために使用するライト信号として動作します.

▶ DMACK#

DMA転送の開始時に,DMARQに対してホストが返す応答信号です.

▶ DMARQ

DMA転送時にデバイス側のデータ転送の準備が整ったときに,デバイスがアサートします.データ転送の方向はDIOR#/DIOW#で制御されます.また,DMACK#とともにハンドシェイクを行います.

▶ INTRQ

INTRQは,選択したデバイスがホストコントローラに対して割り込みを要求するのに使用されます.後述するATAレジスタ中のnIENビットをイネーブルに設定し,かつ有効なデバイスが選択されているとき,そのデバイスのINTRQはイネーブルとなり,割り込みをホストに要求することができます.nIENビットがディセーブル,または有効なデバイスが選択されていないときは,この信号はハイインピーダンス状態となります.

▶ IORDY

デバイス側のデータ転送の準備ができていないとき,レジスタアクセスの転送サイクルを延ばすためにデバイス側がネゲートし,ウェイト信号として使います.デバイスがPIO転送のモード3またはそれ以上の高速な転送サイクルのとき,この信号を利用することができます.

▶ PDIAG#

PDIAG#はデバイス1によってアサートされ,デバイス1が自己診断を終えたことをデバイス0に通知します.

▶ RESET#

ホストがデバイスをリセットするときに使用するハードウェアリセット信号です.

▶ CSEL

CSEL信号を使ったデバイスセレクトを行う場合に使用する信号です.

● **ATAのレジスタ**

ATAのレジスタは,Dataレジスタ以外はすべて8ビット幅のレジスタです.**表2**にATAレジスタ一覧を示し,それぞれのレジスタについて簡単に説明します.

▶ Statusレジスタ

読み出し専用のステータスレジスタです.レジスタのアドレスがCommandレジスタと同じなので,ホストがこのアドレスに書き込み動作をすると,Commandレジスタに値が書き込まれます.

BSYビットが1のとき,このレジスタのBSY以外のビットは無効です.BSYビットは常に有効です.ただし,デバイスがスリープモードのときは,このレジスタは無効です.

INTRQ信号がアサートされているときにStatusレジスタを読

み出すことで，INTRQ信号をネゲートさせることができます．割り込み処理の際は，必ず読み出すようにしてください．INTRQ信号がネゲートしてしまうと都合が悪い場合は，次に説明するAlternate Statusレジスタを読み出してください．Alternate Statusレジスタを読み出してもINTRQ信号はネゲートしません．

▶ Alternate Statusレジスタ

読み出し専用のレジスタです．レジスタのアドレスがDevice Controlレジスタと同じなので，ホストがこのアドレスに書き込みをするとDevice Controlレジスタに値が書き込まれます．Statusレジスタと Alternate Statusレジスタの違いですが，INTRQ信号がアサートされているときにStatusレジスタを読み出すとINTRQ信号がネゲートされるのに対し，Alternate Statusレジスタは読み出してもINTRQ信号に変化を与えません．

▶ Commandレジスタ

書き込み専用のレジスタです．レジスタのアドレスがStatusレジスタと同じなので，ホストがこのアドレスを読み出すとStatusレジスタの内容が読み出されます．このCommandレジスタにコマンドを書く前には，BSYビットとDRQビットが両方とも0でかつ，DMACK#をアサートしていないことを確認してからコマンドを書き込みます．例外としてDEVICE RESETコマンドのみ，BSYビットとDRQビットの確認を必要としません．

コマンドを書き込むと，ただちにコマンドを実行します．また，INTRQがアサートされているときにコマンドを発行するとINTRQはネゲートします．

コマンドを発行する手順としては，バスがアイドル状態のときにデバイスを選択し，選択したデバイスもアイドル状態であるのを確認した後，実行しようとしているコマンドが必要とするパラメータを各ATAレジスタにセットし，最後にCommandレジスタにコマンド値を書き込みます．

▶ Cylinder Highレジスタ

読み書き可能なレジスタです．このレジスタはBSYビットとDRQビットが両方とも0で，かつDMACK#がアサートされていないときに書き込み可能です．

このレジスタは，ATAデバイスに対してCHS方式でアクセスするときにはシリンダの上位バイトの設定として機能し，LBA方式でアクセスするときにはLBA[15：8]として機能します．さらにPACKETコマンドのときには，Byte Count MSBレジスタとして機能します．

▶ Cylinder Lowレジスタ

読み書き可能なレジスタです．このレジスタはBSYビットとDRQビットが両方とも0でかつ，DMACK#がアサートされていないときに書き込み可能です．

このレジスタは，ATAデバイスに対してCHS方式でアクセスするときにはシリンダの下位バイトの設定として機能し，LBA方式でアクセスするときにはLBA[7：0]として機能します．さらにPACKETコマンドのときには，Byte Count LSBレジスタとして機能します．

▶ Device Controlレジスタ

書き込み専用のレジスタです．レジスタのアドレスがAlternate Statusレジスタと同じなので，ホストがこのアドレスを読み出すとAlternate Statusレジスタの値が読み出されます．Device Controlレジスタへの書き込みは，DMACK#がネゲートしているときに可能です．

このレジスタは接続されたデバイスのソフトウェアリセットと，INTRQ信号のイネーブル/ディセーブルの設定に使用されます．このレジスタへの書き込みは，マスタ/スレーブの両デバイスに対して行われるため，SRSTビットに1を書き込むと両デバイスのソフトウェアリセットが実行されます．また，nIENビットの設定は両デバイスに対してINTRQ信号のイネーブル/ディセーブルの設定となります．

▶ Device/Headレジスタ

読み書き可能なレジスタです．このレジスタはBSYビットとDRQビットが両方とも0で，かつDMACK#がアサートされていないときに書き込み可能です．

Device/HeadレジスタのDEVビットは，デバイスを選択するために使用します．0でデバイス0(マスタ)，1でデバイス1(スレーブ)を選択します．また，ATAコマンドの違いでビット定義が異なる点に注意が必要です．たとえば，READ SECTOR(S)コマンドは，ビット6にLBAビットを定義し，セクタのアドレッシング方法をLBA方式で行うか，またはCHS方式で行うかを選択し，ビット3～0にLBAのLBA[27：24]，またはCHSのHEAD値を設定します．

▶ Errorレジスタ

読み出し専用のレジスタです．レジスタのアドレスがFeaturesレジスタと同じなので，ホストがこのアドレスに書き込み動作をするとFeaturesレジスタに値が書き込まれます．BSYビットが0でDRQビットが0，さらにERRビットが1のときにこのレジスタの読み込み値は有効です．また，デバイスがスリープモードのとき，このレジスタは無効です．

Errorレジスタは，パワーオン，ハードウェアリセット，ソフトウェアリセット時，またはEXECUTE DEVICE DIAGNOSTICコマンド，DEVICE RESETコマンドが完了したとき，診断結果をErrorレジスタに反映します．

▶ Featuresレジスタ

書き込み専用のレジスタです．レジスタのアドレスがErrorレジスタと同じなので，ホストがこのアドレスを読み出すとErrorレジスタの内容が読み出されます．BSYビットが0でDRQビットが0でかつ，DMACK#がアサートされていないときにこのレジスタへ書き込みができます．

▶ Sector Countレジスタ

読み書き可能なレジスタです．このレジスタはBSYビットとDRQビットが両方とも0でかつ，DMACK#がアサートされていないときに書き込み可能です．BSYビットまたはDRQビットのどちらかが1のときにこのレジスタを読み出しても，その値は

有効ではありません．また，デバイスがスリープモードのとき，このレジスタは無効です．

▶ Sector Number レジスタ

読み書き可能なレジスタです．このレジスタは BSY ビットと DRQ ビットが両方とも 0 で，かつ DMACK# がアサートされていないときに書き込み可能です．

▶レジスタ

PIO データ転送時に使用するデータレジスタです．DRQ ビットが 1 で，かつ DMACK# がアサートされていないときにアクセス可能です．このレジスタは 16 ビット幅です．接続されている ATA デバイスがコンパクトフラッシュ（CFA デバイス）で，その転送モードが 8 ビット幅の PIO 転送のときは，DD[7：0]が使用されます．

● PIO 転送モード時のバスの動作

PIO 転送モードの場合，ホストからは CS[1：0]#，DIOR#，DIOW# などの信号を使って，ATA レジスタにアクセスを行います．高速な PIO 転送モード 3/4 では，フロー制御用の信号である IORDY 信号をウェイト信号として使用し，確実な転送を行えるようになっています．

これらの信号を用いて，CPU がデータレジスタ経由でデバイスとのデータ転送を行うことを PIO 転送といいます．PIO 転送は，CPU が 1 バイトまたは 1 ワードごとにデータレジスタを読み書きすることで実現します．

レジスタアクセスと CPU によるデータ転送は，基本的にはどちらも同じ PIO 転送ですが，細かなタイミングに違いがあります．

図 1 に PIO 転送によるレジスタアクセスのタイミング波形を，**表 3** にタイミングを示します．

PIO 転送は CPU 主導の転送です．よって，デバイスがホストに対してウェイトを要求したいときは，IORDY をネゲートします．ただし，PIO 転送のモード 0 ～ 2 ではフロー制御はオプションですが，モード 3 ～ 4 では必須となっています．

▶レジスタの読み出しまたはデバイスからデータを読み出すとき

(1) ホストは CS[1：0]# をアサートし，DA[2：0]でレジスタアドレスを確定する

(2) ホストは t_1 後に DIOR# をアサートし，デバイスはアドレスをラッチする

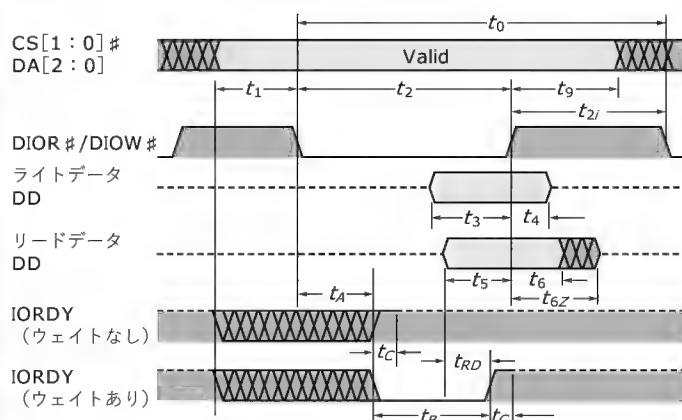
(3) デバイスは，デバイス側のデータ転送準備ができていないときに IORDY をアサートし，処理を待たせる

(4) レジスタアクセスのとき，デバイスは DD[7：0]をドライブしてデータをセットする．PIO データ転送のとき，転送ビット幅の設定にしたがって DD[7：0]または DD[15：0]をドライブし，データをセットする

(5) ホストはデータを読み出し，DIOR# をネゲートする

(6) デバイスは t_6 までデータを保持し，t_{6Z} まで DD[7：0]または DD[15：0]の信号ドライブを開放する

(7) ホストは CS[1：0]# をネゲートする．また同時に，ホストは DA[2：0]を不定にすることができる

〔図 1〕ATA レジスタのアクセスタイミング波形

〔表 3〕ATA レジスタのアクセスタイミング表

記号	パラメータ		モード 0	モード 1	モード 2	モード 3	モード 4
t_0	サイクル時間（min）	8 ビット	600	383	330	180	120
		16 ビット	600	383	240	180	120
t_1	アドレスセットアップ時間（min）		70	50	30	30	25
t_2	DIOR#/DIOW# パルス幅（min）	8 ビット	290	290	290	80	70
		16 ビット	165	125	100	80	70
t_{2i}	DIOR#/DIOW# リカバリ時間（min）		–	–	–	70	25
t_3	DIOW# データセットアップ時間（min）		60	45	30	30	20
t_4	DIOW# データホールド時間（min）		30	20	15	10	10
t_5	DIOR# データセットアップ時間（min）		50	35	20	20	20
t_6	DIOR# データホールド時間（min）		5	5	5	5	5
t_{6Z}	DIOR#3 ステート遅延時間（max）		30	30	30	30	30
t_9	アドレスホールド時間（min）		20	15	10	10	10
t_{RD}	IORDY リードデータ有効時間（min）注		0	0	0	0	0
t_A	IORDY セットアップ時間（min）注		35	35	35	35	35
t_B	IORDY パルス幅（max）注		1250	1250	1250	1250	1250
t_C	IORDY のネゲートからハイインピーダンスまでの時間（max）		5	5	5	5	5

注：モード 0 ～ 2 では IORDY はオプション，モード 3 ～ 4 では IORDY は必須

▶レジスタの書き込みまたはデバイスへデータを書き込むとき

(1) ホストはCS[1:0]#をアサートし，DA[2:0]でレジスタアドレスを確定する
(2) ホストはt_1後にDIOW#をアサートし，デバイスはアドレスをラッチする
(3) デバイスは，デバイス側のデータ転送準備ができていないときにIORDYをアサートし，処理を待たせる
(4) レジスタアクセスのとき，ホストはDD[7:0]をドライブしデータをセットする．PIOデータ転送のとき，転送ビット幅の設定にしたがって，DD[7:0]またはDD[15:0]をドライブし，データをセットする
(5) ホストはDIOW#をネゲートする
(6) ホストはt_4までデータを保持し，DD[7:0]またはDD[15:0]を開放する
(7) ホストはCS[1:0]#をネゲートする．また，同時にホストはDA[2:0]を不定にすることができる

● ホスト側はただのアドレスデコーダ?!

さて，ATAにはいろいろレジスタが割り当てられているようですが，じつはホストコントローラそのものは，これらのレジスタの中身を理解する必要はまったくありません．

たとえば，CPUがSector Numberレジスタに1を書き込もうとしている場合，アドレスバスにはSector Numberレジスタを割り当てたアドレスが，そしてデータバスには1が出力されます．それを受けてATAホストホストコントローラはATA上に，Sector Numberレジスタのアドレス(CS1#="H"，CS0#="L"，DA[2:0]="011")とデータ(DD[7:0]=01h)をそのまま出力，いや横流し(?)するだけです．

逆に，CPUがStatusレジスタを読もうとしている場合は，アドレスバスにStatusレジスタを割り当てたアドレスが出力され，リード信号が出力されます．それを受けて，ATAホストコントローラはATA上に，Statusレジスタのアドレス(CS1#="H"，CS0#="L"，DA[2:0]="111")とリードクロックを出力します．すると，デバイスはステータスレジスタを読み出してATAデータバス上に値を出力するので，ATAホストコントローラはそれを受け取り，ホストCPU側のデータバスに出力します．

つまり，やっていることはアドレスデコーダそのもので，特別何かをコントロールしなければならないというものではありません．そういう意味では，ホストコントローラと呼ぶのもおこがましいくらいです．

● ATAとPCIとのブリッジ

ATAの信号を見ると，一般的な周辺コントローラと同様，チップセレクトやアドレス/データバスがあり，リード/ライトクロックによって行います．しかし，必ずしもそれが実装するプラットホームやホストCPUのバス仕様と一致するとは限りません．今回想定しているのはPCIバスです．32ビット幅かつマルチプレクスバスであるPCIバスに，基本的には16ビットバスであるATAを接続するには，これらの間を適切に変換するコントローラが必要になります．

> **① コラム**
> **x86系CPUとバスアーキテクチャ**
>
> じつはx86系CPUは，奇数アドレスから16ビットサイズや32ビットサイズでのアクセス命令を実行可能です．たとえばメモリアドレスFFFF_0001hにから32ビットサイズのメモリアクセス命令を実行すると，FFFF_0001h/FFFF_0002h/FFFF_0003h/FFFF_0004hの合計4バイトのメモリをアクセスできます．しかしこれは，アセンブラの命令として1命令で実行できたというだけで，実際のPCIバス上でのメモリアクセスは2回のバスアクセスが発生しています．PCIバスは32ビット幅で，FFFF_0001hからの4バイトは4バイト境界を越えているからです．
>
> またx86系CPU以外の32ビットRISC CPUでは，このようなアクセスはバスエラー例外となり，アクセスできないアーキテクチャがほとんどです．今回のシステムのCPUであるSH-4もそうです．
>
> x86系CPUではこのようなアクセスが可能とはいえ，あまり気持ちの良いものではありません．システムのバス幅が32ビットであれば4バイト境界で，64ビットであれば8バイト境界でメモリやI/Oを割り当てたいものです．

● ATAのタイミングや電気的仕様を守る

参考文献1)にもあるように，ATAの各レジスタのアクセスにはモードによりさまざまな規定があります．ATAホストコントローラの仕事は，各モード別に規定されているこれらのタイミングを守ってATAを制御することにあります．

たとえばPIO転送モード4で動いているとき，CPUからあるATAレジスタに書き込み動作が行われたときは，DIOW#の最小パルス幅である70nsを満足させなければなりません．

またATAは5V TTLのバスです．しかしプラットホームによっては信号電圧が3.3Vだったり，それよりさらに低いシステムかもしれません．このような場合には，プラットホームの信号と5V系のATAの信号を接続するためのバスバッファが必要です．ATAホストコントローラは，この電気的な役目も担う必要があります．

2 32ビットPCIバスとATAホストの仕様考察

● ATアーキテクチャとの互換性

ATアーキテクチャのATAレジスタのI/Oアドレスを**表4**に示します．今回，PCIバス上にATAインターフェースを実装し，最終的にはSH-4から制御させるので，PC/AT互換機のIDEと完全互換にする必要はありません．とはいえ，ATAレジスタ群が配置されるアドレス空間や使用する割り込みラインが異なるだけで，それ以外はできるだけ互換性を確保する方向で

仕様を決めたいと思います．

● ATAレジスタの割り当て

第3章で設計したPCIホストコントローラは，もっとも一般的な32ビット33MHzのPCIバスに対応したものです．ATAレジスタの説明でわかるように，ATAの制御レジスタは，データレジスタ以外はすべて8ビットサイズのレジスタです．また，データレジスタは16ビット幅になります．いずれにせよ，PCIバスとはバス幅が異なるので，この間をつなぐにはそれなりのしかけが必要になります．まずはATAレジスタをどうマッピングするかを考えます．

異なるビット幅のバスをもっとも簡単に接続する方法としては，**図2(a)**のように，バス幅の広いほうのデータバスの下位側のビットにそろえて狭いほうのデータバスを並べ，広いバスのアドレスバスをシフトして狭いバスのアドレスバスに接続する方法があります．各レジスタのアドレスが飛び飛びのアドレスに割り当てられるので，実際に使用するアドレス空間より大きなマッピング空間が必要になります．もっともATAの場合は，それほど広大なアドレス空間を必要とするわけではないので，この方法でもプラットホーム側のアドレス空間が足りなくなってしまうことはないでしょう．また，ハードウェアとしてはもっとも簡単な構成になります．

各レジスタの並びを飛び飛びのアドレスではなく連続したアドレスにしたい，または割り当てるアドレス空間を節約したいという場合には，**図2(b)**に示すような方法もあります．バス幅の広いほうのデータバスの中から，アクセスのあるレジスタに該当するバイトだけを切り出して狭いバス側に接続し，アドレスバスはそのままA0をA0へ，A1をA1に接続します．アクセスアドレスによってデータバスの選択/切り替えが必要になるので，多少ハードウェアは複雑になります．

今回は，ATアーキテクチャのIDEと互換性はないものの，少しでも仕様を近くしたいということから，各レジスタを連続したアドレスに並べられる**図2(b)**の方法を採用しました．

● 400nsウェイトレジスタ

ATAを制御するソフトウェアでは，デバイスセレクションやコマンド発行直後などに「400ns以上待つ」という部分が数多く出てきます．今回設計するシステムはCPUとしてSH-4を搭載していますが，SH7750Sでは200MHz，SH7750Rでは240MHzというようにクロック周波数が異なる場合もあります．「400ns待つのにソフトウェアループをn回実行してください」では待ち時間を保証できません．

そこで自前で400nsのウェイト発生回路を実装します．ウェイト発生回路と呼ぶと大げさなようですが，ATAの制御とは何の関係もないダミーのレジスタを用意して，そのレジスタのアクセスに400ns程度の時間をかければよいわけです．

● PIO転送モード設定レジスタ

もう一つ忘れてはならないレジスタが存在します．PIO転送の各種モードを設定するレジスタです．じつはATAの仕様では，ATAデバイスに対して転送モードを設定するコマンドの規定はあるものの，ホストそのものの転送モードの設定方法には規定がありません．これはプラットホームへの実装設計者に任されているのです．

このレジスタも，ATAレジスタや400nsウェイト用のダ

〔表4〕ATアーキテクチャのATAレジスタのI/Oアドレス

プライマリ/セカンダリ	レジスタ群	I/Oアドレス	ビット幅	レジスタ名称
プライマリ	コマンドブロックレジスタ	01F0h	16	Dataレジスタ(R/W)
		01F1h	8	Errorレジスタ(R)/ Featuresレジスタ(W)
		01F2h	8	Sector Countレジスタ(R/W)
		01F3h	8	Sector Numberレジスタ(R/W)
		01F4h	8	Cylinder Lowレジスタ(R/W)
		01F5h	8	Cylinder Highレジスタ(R/W)
		01F6h	8	Device/Headレジスタ(R/W)
		01F7h	8	Statusレジスタ(R)/ Commandレジスタ(W)
	コントロールブロックレジスタ	03F6h	8	Alternate Statusレジスタ(R)/ Device Controlレジスタ(W)
セカンダリ	コマンドブロックレジスタ	0170h	16	Dataレジスタ(R/W)
		0171h	8	Errorレジスタ(R)/ Featuresレジスタ(W)
		0172h	8	Sector Countレジスタ(R/W)
		0173h	8	Sector Numberレジスタ(R/W)
		0174h	8	Cylinder Lowレジスタ(R/W)
		0175h	8	Cylinder Highレジスタ(R/W)
		0176h	8	Device/Headレジスタ(R/W)
		0177h	8	Statusレジスタ(R)
	コントロールブロックレジスタ	0376h	8	Alternate Statusレジスタ(R)/ Device Controlレジスタ(W)

〔表5〕設計したATAインターフェースのI/Oアドレス割り当て

レジスタ群	オフセット	ビット幅	レジスタ名称(リード/ライト)
コマンドブロックレジスタ	xx00h	16	Dataレジスタ(R/W)
	xx01h	8	Errorレジスタ(R)/ Featuresレジスタ(W)
	xx02h	8	Sector Countレジスタ(R/W)
	xx03h	8	Sector Numberレジスタ(R/W)
	xx04h	8	Cylinder Lowレジスタ(R/W)
	xx05h	8	Cylinder Highレジスタ(R/W)
	xx06h	8	Device/Headレジスタ(R/W)
	xx07h	8	Statusレジスタ(R)/ Commandレジスタ(W)
コントロールブロックレジスタ	xx08h ～ xx0Dh	― ―	未使用
	xx0Eh	8	Alternate Statusレジスタ(R)/ Device Controlレジスタ(W)
	xx0Fh	―	未使用
ウェイト用レジスタ	xx10h	32	400nsウェイト専用ダミーレジスタ(R)
	xx14h ～ xx1Fh	― ―	上記レジスタのイメージ

ミーレジスタと同じ空間に割り当てても何ら問題ありませんが，今回設計するのはPCIデバイスなので，PCIデバイスのコンフィグレーション空間内に実装することにしましょう．

PIO転送モードの設定は，初期化のときに一度だけ設定すれば，基本的にはそれ以降変更する必要はありません．使用頻度の少ない，初期設定に関するレジスタであるといえます．そういう意味でも，PCIコンフィグレーションレジスタ空間に割り当てたほうが無難なレジスタかと思います．

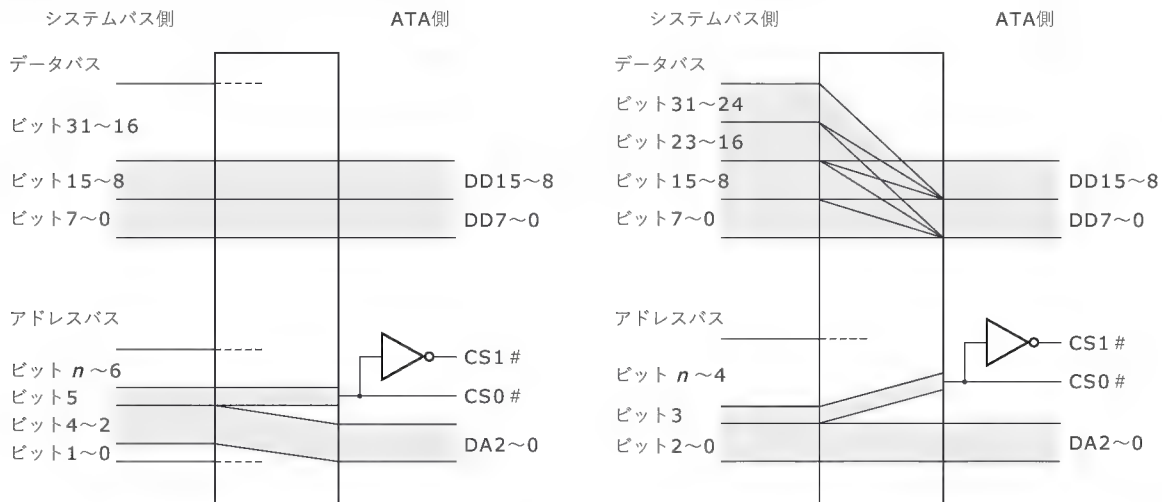
3 ATAインターフェースの仕様の決定

これまで解説してきた内容を踏まえて，これから設計するATAインターフェースの仕様を決定します．

● ベースアドレスレジスタ1本使用

表5に，設計したATAインターフェースのアドレス割り当てを示します．SH-4にI/O空間はありませんが，PCIバスとして今回のようなI/Oインターフェースを実装する場合は，やはりI/O空間に割り当てるのがスマートです．前半16バイトにコマンドブロックレジスタ群とコントロールブロックレジスタ群を，後半に400nsウェイト用のダミーレジスタを割り当て，PCIデバイスとして32バイトのI/O空間を要求することにします．

オフセット+00hのデータレジスタの上位8ビットと，オフセット+01hのErrorレジスタやFeaturesレジスタのアドレスが重なっていて大丈夫かと思われるかもしれませんが，この点についてホストコントローラが感知する必要はありません．デバイス側はアクセスのあるレジスタをきちんと判定して動作します．

〔図2〕バス幅の異なるデバイスのレジスタマッピング例

レジスタ群	I/Oアドレス	ビット幅	レジスタ名称
コマンドブロックレジスタ	xx00h	16	Dataレジスタ(R/W)
	xx04h	8	Errorレジスタ(R)/Featuresレジスタ(W)
	xx08h	8	Sector Countレジスタ(R/W)
	xx0Ch	8	Sector Numberレジスタ(R/W)
	xx10h	8	Cylinder Lowレジスタ(R/W)
	xx14h	8	Cylinder Highレジスタ(R/W)
	xx18h	8	Device/Headレジスタ(R/W)
	xx1Ch	8	Statusレジスタ(R)/Commandレジスタ(W)
コントロールブロックレジスタ	xx20h	－	未使用
	xx24h	－	未使用
	xx28h	－	未使用
	xx2Ch	－	未使用
	xx30h	－	未使用
	xx34h	－	未使用
	xx38h	8	Alternate Statusレジスタ(R)/Device Controlレジスタ(W)
	xx3Ch	－	未使用

(a) 32ビットバス中の下位8/16ビットを使ってAddress[4:2]→DA[2:0]に配線する

レジスタ群	I/Oアドレス	ビット幅	レジスタ名称
コマンドブロックレジスタ	xxx0h	16	Dataレジスタ(R/W)
	xxx1h	8	Errorレジスタ(R)/Featuresレジスタ(W)
	xxx2h	8	Sector Countレジスタ(R/W)
	xxx3h	8	Sector Numberレジスタ(R/W)
	xxx4h	8	Cylinder Lowレジスタ(R/W)
	xxx5h	8	Cylinder Highレジスタ(R/W)
	xxx6h	8	Device/Headレジスタ(R/W)
	xxx7h	8	Statusレジスタ(R)/Commandレジスタ(W)
コントロールブロックレジスタ	xxx8h	－	未使用
	xxx9h	－	未使用
	xxxAh	－	未使用
	xxxBh	－	未使用
	xxxCh	－	未使用
	xxxDh	－	未使用
	xxxEh	8	Alternate Statusレジスタ(R)/Device Controlレジスタ(W)
	xxxFh	－	未使用

(b) 32ビットバス中の各8ビットを切り替えてAddress[2:0]→DA[2:0]に配線する

また，オフセット +08h から 4 バイト境界をまたがないアドレスに未使用空間があるので，ダミーレジスタをそこに実装したほうが，使用する空間を半分にできるという意見もありそうです．たしかにそのとおりですが，オフセット +00h から +0Fh まではそのまま機械的に ATA 側にデコードするだけにしたいので，途中に用途の異なるレジスタを入れたくないため，このようにしました（後述するようにデバイス内部のシーケンサのステートを，各 ATA レジスタ制御用とダミーレジスタ用で分けたため）．

ちなみに，じつはこのアドレスマップ仕様は，PC カード ATA のカードコンフィグレーションインデックス 0 の場合のアドレス割り付けにも近いものがあります．

● **PCI コンフィグレーションレジスタ仕様**

PCI デバイスなので，当然ベンダ ID とデバイス ID が必要になります．ここでは ID として筆者が所属する会社のベンダ ID と，今回の設計用にデバイス ID を割り当てました．

ステータスレジスタの DEVSEL# 応答ビットは中速応答の設定で，またシステムエラーやパリティエラーの検出は今回は省略したので，その他のステータスレジスタのビットはすべて‘0’にします．コマンドレジスタには I/O 空間イネーブルビットを実装します．

PCI デバイスのクラスコードには，ストレージクラスに IDE コントローラという分類もあります．しかし，このクラスコードを見て，この ATA インターフェースが AT アーキテクチャ標準の IDE と完全互換であると認識されると困るので，今回はストレージクラスの中のその他のデバイスを選択しました．

リビジョン ID にはとりあえず 01h を，ヘッダタイプはブリッジチップではない標準の PCI デバイスなので 00h とします．

〔表 6〕設計した ATA インターフェースの PCI コンフィグレーションレジスタ

オフセット	ビット幅	レジスタ名称	値	リード/ライト
+00h	16	ベンダ ID	6809h	R
+02h	16	デバイス ID	8004h	R
+04h	16	コマンドレジスタ ビット 0　I/O イネーブルビット	–	R/W
+06h	16	ステータスレジスタ ビット 10/9　DEVSEL 応答	01b	R
+08h	8	リビジョン ID	01h	R
+09h	8	プログラムインターフェース	00h	R
+0Ah	8	サブクラスコード	80h	R
+0Bh	8	ベースクラスコード	01h	R
+0Eh	8	ヘッダタイプ	00h	R
+10h	32	ベースアドレスレジスタ 0 ビット 15～5　32 バイト要求 ビット 0　I/O 空間	– 1b	R/W R
+3Ch	8	インタラプトライン ビット 7～0	–	R/W
+3Dh	8	インタラプトピン	01h	R
+40h	32	PIO 転送モードレジスタ ビット 2～0	–	R/W

注：表記の内レジスタ/ビットはライト時は無視/リード時は 0
R/W レジスタは PCI バスリセット時にクリア

ベースアドレスレジスタは 1 本のみ使用するので，ベースアドレスレジスタ 0 に実装し，残りのベースアドレスレジスタは未使用です．

また，割り込み制御も必要になるので，インタラプトピンレジスタには INTA# の使用を明示し，インタラプトラインには 8 ビットのリード/ライトレジスタを実装しておきます．

それ以外のレジスタは使用しないので，ライト時は無視，リード時は 00h を返すように設計します．

● **PCI コンフィグレーションデバイス固有レジスタ**

もう一つ，PIO 転送のホスト側のモードを設定/保持するレジスタも必要です．このレジスタは PCI のコンフィグレーションレジスタ空間に実装することにしたので，デバイス固有のレジスタを配置できる，40h 以降のアドレスにマッピングしなければなりません．

今回はこれ以外に独自のレジスタを実装する必要はないので，とくに配置アドレスを考えず，40h に PIO 転送モード設定レジスタを実装しました．

以上をまとめて，**表 6** に PCI コンフィグレーションレジスタを示します．

4 ATA インターフェースの設計

● **PCI バスクロックによるシーケンサ**

今回の設計では，タイミングはすべて PCI バスクロックを 33MHz（30ns）を基準に設計しています．

ここで設計する PCI デバイスは I/O 空間を使用するので，ローカルバスシーケンサ内には，PCI デバイスとして必要なコンフィグレーション空間の制御と，ATA レジスタにアクセスするための I/O 空間の制御を記述します．

今回の PCI デバイスの設計では，参考文献 1）の TECH I Vol.3『PCI デバイス設計入門』に掲載されている，PCI デバイス設計の各種 HDL ソースを参考にさせていただきました．PCI ターゲットシーケンサの部分などは，この本の HDL ソースをほとんどそのまま流用させていただいています．筆者の方々にお礼申し上げます．

● **PCI ターゲットシーケンサ**

参考文献 1）の設計データの中からベースにしたのは，第 8 章のディジタル I/O ボード & FIFO 搭載 I/O ボードの設計に解説のある `PIO.VHD` です．ベースアドレスレジスタを 1 本，そして割り込み INTA# だけを使った PCI デバイスです．この設計のシーケンサのうち，PCI バス側を制御する PCI ターゲットシーケンサには，いっさいの変更を加えていません．

コンフィグレーションレジスタにはベンダ ID やベースアドレスレジスタのビット幅など，多少仕様の違いがあるので，それらの定義部分などは変更しています．

● **ATA インターフェース回路**

図 3 に ATA インターフェースの回路を示します．実際のデバ

イスとしてはCPLD/FPGAを使っているので，実際のピン番号は配線のしやすさなどを考慮して適当に決めてください．

なお，ATAのデータバスはプルアップ抵抗を実装していないので，非アクセス時はホスト側が信号をドライブするようにしています．

● RESET，INTRQ，DMA信号

リスト1(章末)に設計したシーケンサのVHDLソースを示します．

INTRQは正論理の割り込み信号なので，反転してINTA#に出力します．またRESET端子はPCIバスのRST#をそのまま出力したので，PCIバスリセット時にハードウェアリセットがかかります．ソフトウェアでは制御できません．

なお，今回のATAインターフェースはPIO転送専用なので，DMARQやDMACK#は使用しません．出力信号であるDMACK#は常時"H"レベルを出力し，入力信号であるDMARQは入力ピンだけ定義し，内部では使用していません．

● 各ATAレジスタのアクセス

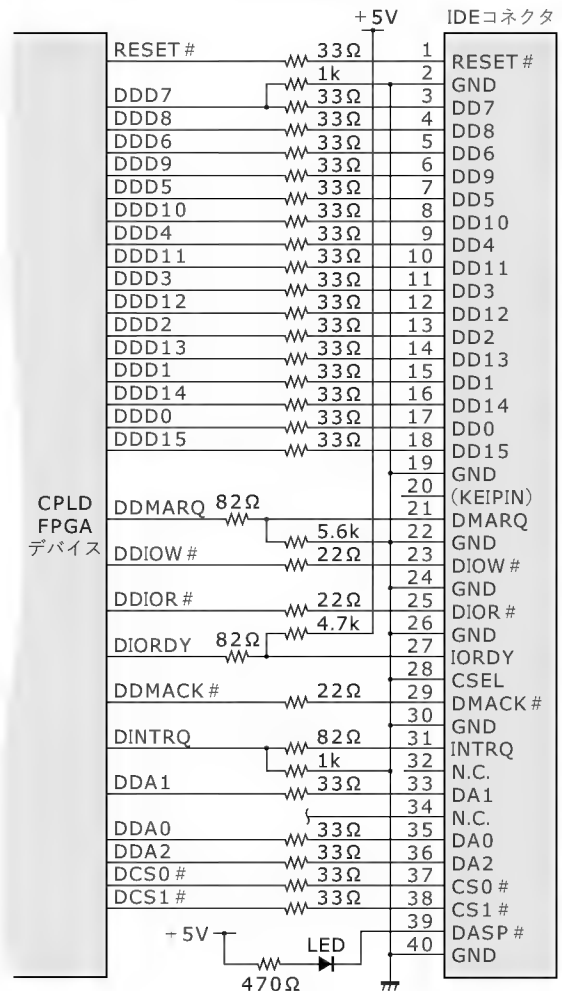
ATAレジスタのうち，データレジスタは16ビット，それ以外はすべて8ビットレジスタです．しかしPCIバスは32ビット幅なので，32ビット幅で一度にアクセスが発生するかもしれません．

PCIバスの規格では，禁止されている領域やサイズでアクセスがあった場合，ターゲットアボートなどでイニシエータ側にエラーを通知させるしくみが用意されています．しかし，ここでは特にエラーは発生させずに，アドレスバスに出力されているアドレスに該当するレジスタがアクセスされたものとみなして，アクセス先のレジスタとアクセスサイズを決め打ちして処理をします．もっとも，これが通用するのはPCIバスではI/O空間に限られます．

たとえば，今回のATAインターフェースの設計では，もしDataレジスタに32ビットサイズでアクセスがあっても，Dataレジスタのサイズは16ビットなので，Dataレジスタにのみアクセスがあったものとみなして下位16ビットを取り出して処理します．同様にCylinder Lowレジスタに16/32ビットサイズでアクセスがあっても，Cylinder Lowレジスタのサイズは8ビットなので，Cylinder Lowレジスタに下位8ビットを取り出して処理します．さらに同様にSector CountレジスタやDevice/Headレジスタに16ビットサイズでアクセスがあっても，これらのレジスタのサイズは8ビットなので，ビット23～16の8ビットだけを取り出して処理します．

● アクセスタイミング制御

各ATAレジスタのアクセスには，各PIO転送モードにより規定のある制御タイミングを満足するように，ウェイトカウンタを用意してアクセス速度を確保します．**図1**と**表3**を参照してください．ATAの制御で注目が必要なタイミングは，アドレスセットアップとDIOR#/DIOW#の最小パルス幅，そしてサイクルタイムです．

〔図3〕ATAインターフェースの回路図

たとえば一般的なSRAMでは，アドレスセットアップタイムといった場合にはリード/ライトクロックの立ち上がりの何ns前までに値が確定していなければならない，というパラメータです．しかし，**図1**を見るとわかるように，ATAの場合のアドレスセットアップタイムは，DIOR#/DIOW#の立ち下りまでの確定時間と規定されています．ただし，データセットアップタイムはDIOR#/DIOW#の立ち上りまでの時間で規定されています．

また，サイクルタイムも重要です．セットアップタイムとリード/ライトクロックの最小パルス幅，そして場合によってはホールドタイムを合計して全体のアクセス時間となりますが，連続してアクセスする場合は，アクセスとアクセスの間にウェイトが必要なものがあります．これをサイクルタイムとして，直前のアクセス開始時点から次のアクセス開始までの最小時間で規定されています(しかし，モード0で600nsというのは……遅い！)．

さらによく見ると，モード0から2までは8ビットレジスタに対するアクセスと，16ビットデータレジスタに対するアクセスとで，若干タイミングが異なるところがあります．とくにモード2のDIOR#/DIOW#の最小パルス幅は3倍近い差になっていま

す．とはいえ，細かく対応すると制御がたいへんなので，ここは涙をのんで遅いほうに合わせることにします．

以上のタイミングを，今回のシーケンサ（基準クロック30ns）で制御する場合の各クロック数を**表7**に示します．最小パルス幅が70nsという場合，あと10ns短ければ2クロックで済んだのに……などと考えてしまいますが，規定時間を満足させるために3クロックとしています．

● **サイクルタイムの考慮**

表7中のサイクルタイムウェイト数や*PCIwait*というパラメータは何でしょうか．

今回参考にしたシーケンサの基本動作波形を**図4**に示します．このPCIターゲットシーケンサは，PCIバス側のタイミングを制御するPCIターゲットシーケンサと，ローカルバス側のタイミングを制御するローカルバスシーケンサに分かれていて，それぞれPCIバスクロックに同期して動いています．そしてPCIターゲットシーケンサからローカル側のスタート信号が出力されると，1クロック遅れてローカルバスシーケンサが動き始め，ローカルバス側での制御が終わった信号が出力されると，1クロック遅れてPCIターゲットシーケンサがそれを認識します．このように，制御方法としては非常に簡単で理解しやすいシーケンサなのですが，片方の処理が終わるまで反対側が待つという，パイプライン制御されていないシーケンサのため，全体の処理が終わるまでに時間がかかります．

よって，実際にローカルバス，つまりATAレジスタへのアクセスの前に3クロック程度，後に3クロック程度の時間がかかります．このためATAレジスタへ連続してアクセスするという場合，直前のアクセスの後に6クロック以上の間が空きます．表中の*PCIwait*＝6というのは，そのクロック数です．

よって，モード1から3まではアドレスセットアップと最小クロック幅の合計クロック数より，サイクルタイムのクロック数が大きいにもかかわらず，サイクルタイムを確保するためのウェイトを入れなくても，自動的に（？）PCIバス側のウェイトが加算されることになるので，サイクルタイムを確保するためのウェイトが不要になりました．

モード4の場合は，最小パルス幅が70nsなので3クロックにしなければならず，セットアップの1クロックと合わせて4クロックになり，結局のところATAレジスタのアクセス制御だけで120nsの時間がかかるので，PCIバス側のウェイトがなくてもサイクルタイムを満たすことができます．

● **サイクルタイムウェイトの挿入方法**

結局，モード0ではサイクルタイムを満たすために1クロックのウェイトが必要になります．

さて，サイクルタイムウェイトはどのように入れればよいのでしょうか．サイクルタイムとは連続してアクセスされた場合に間を空けるためのもの……ですから，もっとも簡単にウェイトを挿入するには，すべてのアクセスにウェイトを加算するという方法があります．たしかにこれで連続アクセスの場合，サイクルタイムを保証することができますが，アクセスとアクセスの間にCPUが別の処理をする場合でも，それぞれのアクセスにサイクルタイムを保証するためのウェイトが入ってしまうので，パフォーマンスが上がりません．

そこで，直前のアクセス完了からサイクルタイムウェイトをカウントアップまたはカウントダウンし，次にアクセスが発生した場合，規定の値まで達していなかったらCPU側をウェイト状態にして，必要なウェイト時間を確保してから実際に次のアクセスを開始するという方法があります．これにより，連続してアクセスが発生した場合でもサイクルタイムが保証され，またアクセスとアクセスの間にCPUが別の処理をする場合にはその間にサイクルタイム分の時間が経過し，次にアクセスするときはウェイトが挿入されずにアクセスを行うことが可能になります．

● **各タイミング制御**

今回のシーケンサでは，アドレスセットアップウェイトとDIOR#/DIOW#最小パルス幅ウェイトの二つのウェイト規定変数（`SETUP_Count`/`WIDTH_Count`）と，一つのウェイトカウンタ（`WAIT_Count`）を使ってタイミングを制御しています．

ライトデータのセットアップはDIOW#の立ち上がりまでの時間で規定されています．これはDIOR#/DIOW#の最小パルス幅よりも十分に短いので，DIOW#をアサートするタイミングでATAのデータバスに出力しています．

リードに関しては最速のモード4でも20nsのセットアップ時間が確保されているので，PCIデバイスとして使えるCPLD/FPGAであれば十分満足できる時間です．シーケンサ内ではDIOR#を立ち上げるときに読み込んでいます．

● **ホールドタイムなど**

セットアップタイムだけでなく，ホールドタイムの検証も必要です．

〔**表7**〕**各PIO転送モードとシーケンサ内のクロック数**

モード	アドレスセットアップタイム		データセットアップタイム		DIOR#/DIOW#パルス幅		サイクルタイム		サイクルタイムウェイト数
	時間	クロック数 a	時間	クロック数	時間	クロック数 b	時間	クロック数 c	$c-(a+b+PCIwait)$ ただし，$PCIwait=6$
0	70	3	60	2	290	10	600	20	$20-(3+10+6)=1$
1	50	2	45	2	290	10	383	13	$13-(2+10+6)=-5\rightarrow0$
2	30	1	30	1	290	10	330	13	$13-(1+10+6)=-4\rightarrow0$
3	30	1	30	1	80	3	180	6	$6-(1+3+6)=-4\rightarrow0$
4	25	1	20	1	70	3	120	4	$4-(1+3+6)=-6\rightarrow0$

アドレスホールドタイムについては，DIOW#を“H”レベルに戻した次のステートでCSn#を“H”レベルに戻しているので，最長のモード0(20ns)でも大丈夫です．

書き込み時のデータホールドタイムに関しては，ATAのデータバスに出力した値は，そのまま次のアクセスが発生するまで保持させているので，モード0の30nsでも十分すぎるほどです．

読み出しのときのデータバスの3ステート遅延時間はすべてのモードで30nsです．DIOR#を“H”レベルに戻した次のステートでデータバスの方向を出力に切り替えているので大丈夫です．

● **IORDYによるウェイト制御**

IORDYによるデバイス側からのウェイト要求は，すべてのモードでDIOR#/DIOW#の立ち下がりから35nsと規定されています．今回のシーケンサでは，DIOR#/DIOW#の立ち上げのタイミングでIORDY信号をサンプリングしています．ここでウェイトが要求されていればクロックをアサートしたまま，次のタイミング(30ns後)まで待ちます．よって30ns単位でアクセス時間を加算していきます．

〔図4〕PCIターゲットシーケンサの基本動作波形

● **PIO転送モードレジスタとダミーレジスタ**

PIO転送モードレジスタはPCIコンフィグレーションレジスタ空間に実装しています．ビット2～0に実装しているので，40hを8/16/32ビットのいずれのサイズでもアクセスできます．

また，400nsウェイト用のダミーレジスタは，ATAレジスタの制御とは別のステートを用意し，このステートの前後およびPCI側のウェイトを考慮して6クロックのウェイトをカウントしています．

5 ATA制御プログラムの作成

● **SH-4システムへの移植**

ATA/ATAPIの制御プログラムは，参考文献2)のATA/ATAPI制御プログラムを移植しました．ATA/ATAPIデバイスの初期化方法や，実際のHDDやCD-ROMのアクセス制御方法の詳細はそちらを参照してください．

ここでは，参考文献2)の第5章のATA制御プログラムを，本PCIデバイスのATAインターフェースの仕様に合うように物理層部分を合わせこみます．具体的には，

(1) PCIデバイスの検索とリソースの取得
(2) 割り込み処理の初期化
(3) 転送モード設定の実行

となります．

● **ATAインターフェースデバイスの検索**

通常のPCIデバイスのドライバと同様に，ベンダIDとデバイスIDを指定してPCIデバイスを検索し，割り当てられたI/OアドレスとIRQ番号を取得します．もし同じPCIデバイスが複数存在しても，最初に発見したデバイスのみを使うので，インデックスは0でかまいません．

参考文献2)のサンプルプログラムでは参考文献1)のPCIデバッグライブラリを使用していますが，そこは第3章で作成したSH-4用PCI BIOSを呼び出すように書き替えます．

● **割り込み処理の初期化**

移植したプログラムでは，プライマリIDEの割り込みとしてIRQ14を，セカンダリIDEの割り込みとしてIRQ15を決め打ちで使っていましたが，その部分を取得したIRQ番号に置きかえます．ただし，IRQ番号と呼んでいますが，PC/AT互換機のIRQとは異なり，SH-4システム上での割り込み認識番号として使用しています．

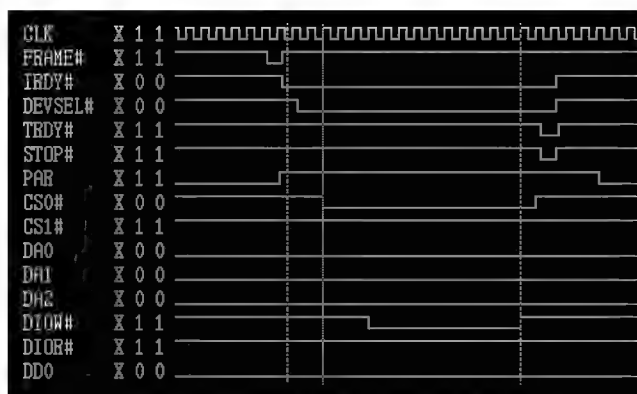
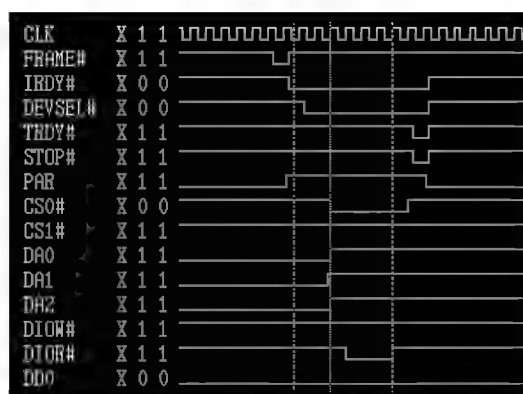
● **転送モードの実行**

ソース中のコメントアウト(実際には関数の頭での`return`文)を削除して，実際にPIO転送モードの設定変更を，ホスト側およびドライブ側両方で実行します．

● **動作確認**

写真1に実際にATAデバイスを制御したときのPCIバスとATAのバスの動作波形を示します．

写真1(a)はPIO転送モード0のときにDataレジスタへ書き込み動作をしたときの波形です．CS0#がアサートされてから3クロック後にDIOW#がアサートされていることがわかります．

〔写真 1〕実際の動作波形

(**a**) PIO 転送モード 0　Data レジスタライト

(**b**) PIO 転送モード 4　Status レジスタリード

また DIOW# のアサート期間は 10 クロックになっています．

写真 1(**b**)は，モード 4 で Status レジスタを読み出したときの波形です．Status レジスタのアドレスが出力され，CS0# のアサートから 1 クロックで DIOR# がアサートされ，アサート期間は 3 クロックになっています．

まとめ

● パフォーマンス向上のための工夫

今回はもっとも簡単な ATA インターフェースを実現するということで，PIO 転送モードのみに対応した ATA を設計しました．ATA レジスタ制御タイミングでも説明したように，今回のシーケンサではローカルバスシーケンサの前後に，PCI ターゲットシーケンサの処理時間があるため，全体としてのアクセス時間が非常に長くなっています．このためパフォーマンス的には非常に悪い ATA インターフェースとなっています．

これを改善するには，シーケンサ全体の最適化/チューニングが必要です．たとえば，ローカルバスシーケンサの起動信号をもっと速い段階で出力したり，シーケンサ間のフラグが 1 クロック送れて認識されるのを考慮した投機的な出力/制御方法を採用するなどです．

● バスマスタ対応の DMA 転送

さらに大幅なパフォーマンスの向上をめざすには，ATA 側をマルチワード DMA 転送に対応させ，PCI 側をバスマスタ対応にすることです．マルチワード DMA 転送では最高でもモード 2 の 16.6M バイト/秒なので，PIO 転送のモード 4 と変わらない……という意見もありそうですが，PIO 転送はデータ転送そのものを CPU で行うので，CPU の使用効率が良くありません．シングルタスクではあまり違わないでしょうが，マルチタスクで使おうとすると，データ転送を DMA 転送で行うほうが，システム全体のパフォーマンスがずっと向上します．

もちろん，さらに Ultra DMA 転送を使うという奥の手がありますが，Ultra DMA 転送ではクロックの立ち上がりと立ち下りの両方のエッジを使うので，設計の難易度はさらに上がります．

今回の記事を参考に，より高速な ATA インターフェースを実現するのは，あなたです．

参考文献

1)『PCI デバイス設計入門』，TECH I Vol.3，CQ 出版（株）
2)『ATA（IDE）/ATAPI の徹底研究』，TECH I Vol.10，CQ 出版（株）

やまたけ・いちろう　来栖川電工有限会社

〔リスト 1〕ATA インターフェースの VHDL ソース（一部）

```
-- ************************************************************************ --
-- ********** ローカルバスシーケンサ
-- ************************************************************************ --
LOCAL BUS Seq : process( PCICLK, RST n )

        ～中　略～

begin

-- ********** リセット時動作 ********** --
    if ( RST n = '0' ) then -- PCI バスリセットがアサートされたとき

            ～中　略～

         -- PIO 転送モードレジスタ
         PIO Mode  <= (others => '0');

         -- ATA/ATAPI 信号初期化
         DD Port <= (others => '0');
```

〔リスト1〕**ATAインターフェースのVHDLソース（一部）（つづき）**

```
        DD HiZ   <= '0';      -- ATAデータバス出力方向
        CS0n     <= '1';
        CS1n     <= '1';
        DIOWn    <= '1';
        DIORn    <= '1';
        DA       <= (others => '0');

        -- ATAレジスタアクセスウェイトクリア
        ATA Count    := '0';
        SETUP Count := (others => '0');
        WIDTH Count := (others => '0');
        WAIT Count  := (others => '0');

    elsif (PCICLK'event and PCICLK = '1') then

    -- PCI INTA#割り込み出力
    if (INTRQ = '1') then     -- ATA割り込み入力
        INTA HiZ <= '0';      -- 割り込み出力
    else
        INTA HiZ <= '1';      -- 割り込み非出力
    end if;


-- ********** ローカルバスシーケンサ ステートマシン ********** --

        LOCAL CURRENT STATE := LOCAL NEXT STATE ;
        case LOCAL CURRENT STATE is


    -- ********** LOCAL IDLE時の動作 ********** --
        when LOCAL IDLE => -- ローカルバスシーケンサ スタート指示待ち

            if (LOCAL Bus Start = '1' ) then  -- ローカルバスシーケンサ スタート!

                if (Hit Config = '1') then    -- コンフィグレーションサイクルヒット
                    LOCAL NEXT STATE := LOCAL CFG ACCESS;  -- コンフィグレーションステートへ
                end if;

                if (Hit Io = '1') then        -- I/Oサイクルヒット

                    if (PCI Address(4) = '1') then     -- 400nsウェイト用I/Oアクセス
                        WAIT Count  := (others => '0');
                        LOCAL NEXT STATE := LOCAL 400n WAIT;

                    else -- ATAレジスタアクセス
                        if (ATA Count = '1') then     -- ATAサイクルタイムウェイト完了
                            if (PCI Address(3) = '0') then   -- ATAコマンドブロックレジスタ空間(0h~7h)
                                CS0n <= '0';
                            else                              -- ATAコントロールブロックレジスタ空間(8h~Fh)
                                CS1n <= '0';
                            end if;
                            DA   <= PCI Address(2 downto 0);
                            if (PCI BusCommand(0) = '0' ) then         -- リードサイクル
                                DD HiZ <= '1'; -- ATA(IDE)データバスドライブ解放
                            end  if;
                            case PIO Mode is    -- PIO転送モード別パラメータ設定
                            when "000" =>  -- モード0
                                SETUP Count := "11";     -- アドレスセットアップウェイト 3クロック
                                WIDTH Count := "1010";   -- クロック幅ウェイト 10クロック
                            when "001" =>  -- モード1
                                SETUP Count := "10";     -- アドレスセットアップウェイト 2クロック
                                WIDTH Count := "1010";   -- クロック幅ウェイト 10クロック
                            when "010" =>  -- モード2
                                SETUP Count := "01";     -- アドレスセットアップウェイト 1クロック
                                WIDTH Count := "1010";   -- クロック幅ウェイト 10クロック
                            when others => -- モード3/4
                                SETUP Count := "01";     -- アドレスセットアップウェイト 1クロック
                                WIDTH Count := "0011";   -- クロック幅ウェイト 3クロック
                            end case;
                            WAIT Count := "0001";    -- ウェイトカウンタクリア(次のステート遷移分で1ウェイトカウント)
                            LOCAL NEXT STATE := LOCAL SETUP WAIT;

                        else -- ATAサイクルウェイトが経過するまでアイドルに待機
                            LOCAL NEXT STATE := LOCAL IDLE;
                        end if;

                    end if;
                end if;
            else -- ローカルバスシーケンサ スタートフラグがまだならこのステートにとどまる
                LOCAL NEXT STATE := LOCAL IDLE;
            end if;
                                                  ～以下省略～
```

エピローグ

まだ足りない！　まだまだこれから！

今後の展開と基板入手方法

井倉将実

● 今後の展開

以上で，グラフィックス出力とキーボード/マウス，そしてHDDやCD-ROMドライブがつながるパソコンを実現できました．しかし機能的には，まだまだ足りないのが現状です．

今回の特集で解説を掲載できなかったインターフェースとしては，PCカード/CFカードコントローラ，光ディジタルIN/OUT対応PCMサウンドカードなどがあります．

また，ATAインターフェースの2チャネル化やシリアルやパラレルポートなど，これまで設計してきた各種インターフェースを1枚にまとめ，ATX形状の基板を起こす計画があります．特集原稿執筆段階では，まだスケジュールが決まっていませんでしたが，本号が発売される頃には，何らかのアナウンスを行えると思います．下記URLを参照ください．

■ **URL**：http://www.kurusugawa-ele.co.jp/

● 基板の入手方法

今回の特集で設計/製作した各種基板，また開発ツールなどの入手方法について説明します．

▶プロセッサボード

プロセッサボードは，BIANCA（ビアンカ）という愛称を付けて呼んでいます．特集執筆時点では，試作基板および量産試作基板が完成した段階で，問題なく動作している状況です．本号が発売される頃にはプロセッサボードの量産版も完成し，ドキュメントなどの準備をしている頃だと思われます．出荷開始状況など最新情報は，次のURLを参照してください．

■ **発売元**：**来栖川電工（有）**
■ **品名**：**SH-4搭載PCI/PICMG対応CPUカード**
■ **型名**：**KEI-BIANCA/SH4**
■ **URL**：http://www.kurusugawa-ele.co.jp/product/system/bianca/

▶グラフィックスボード

今回は，XILINX Spartan-II搭載PCI開発キットのオプションとして，ビデオ用D-Aコンバータを搭載したKEI-EVSP2/VIDEOを使いました．KEI-EVSP2/VIDEOは，すでに販売中のPCI試作評価用ボードおよびオプションのキットです．

■ **発売元**：**来栖川電工（有）**
■ **品名**：**XILINX Spartan-II搭載PCI開発キット/ビデオ出力オプション付き**
■ **型名**：**KEI-EVSP2/VIDEO**
■ **URL**：http://www.kurusugawa-ele.co.jp/product/pci/evsp2/

▶ATAインターフェース

特集執筆時点では基板化は行っていません．ビデオカードのところでも紹介したKEI-EVSP2/150を使い，拡張ピンヘッダの部分を手配線で試作しています．

KEI-EVSP2/150は，すでに販売中のPCI試作評価用ボードです．

■ **発売元**：**来栖川電工（有）**
■ **品名**：**XILINX Spartan-II搭載PCI開発キット**
■ **型名**：**KEI-EVSP2**
■ **URL**：http://www.kurusugawa-ele.co.jp/product/pci/evsp2/

▶PS/2キーボード＆マウスインターフェース

PCIバス接続部分は，上記ATAインターフェースと兼用で設計しました．キーボードやマウスのデータを変換するマイコンには，市販されているM16CカードマイコンOAKS16-BoardKit（オークス電子）を使いました．

M16Cのコンパイラ/デバッガはすべて，OAKS16-BoardKitに添付されているもの，または三菱電機のWebサイトから入手できるフリーのものが使えます．

■ **発売元**：**オークス電子（株）**
■ **品名**：**OAKS16シリーズ**　■ **型名**：**OAKS16-BoardKit**
■ **URL**：http://www.oaks-ele.com/oaks16/index_oaks16.htm
■ **M16C関連URL**：http://www.infomicom.mesc.co.jp/M16C/mctopj.htm

▶PCIバックプレーン

これらのPCIボードをまとめて一つのシステムとするためには，PCIバックプレーンが必要です．今回は市販PCIバックプレーンの一つであるPCM-PCM05（インタフェース）を使いました．

■ **発売元**：**（株）インタフェース**
■ **品名**：**PCIバス5スロットバックプレーン**
■ **型名**：**PCM-PCM05**
■ **URL**：http://www.interface.co.jp/catalog/prdc.asp?name=pcm-pcm05

▶SH-4用開発ツール

SH-4上で動作するプログラムの開発やデバッグは，PARTNER-J（京都マイクロコンピュータ）およびexeGCC（同）を使いました．JTAGデバッガとしてはかなり高速で，SH-4用JTAGデバッガの中では安価に購入できる，コストパフォーマンスの良いデバッガおよびコンパイラだと思います．

■ **発売元**：**京都マイクロコンピュータ（株）**
■ **品名**：**SH-4用JTAGデバッガPARTNER-J**
■ **型名**：**PARTNER-J Model10/SH4**
■ **URL**：http://www.kyoto-microcomputer.co.jp/j/jsh4.htm

▶FPGA設計開発ツール

今回のシステムでは，プロセッサボード，グラフィックスボード，ATAインターフェースボードなど，FPGAとしてすべてSpartan-II（ザイリンクス）を搭載しています．無料版WebPACK ISEには論理合成エンジンも含まれているので，VHDLソースの入力から論理合成，配置配線，ダウンロード/デバイスプログラミングまで，FPGA設計開発ツールは，Spartan-II対応の設計開発ツール一つで十分です．本特集で設計したVHDLソースも，すべて無料版のWebPACK ISEで論理合成/配置配線が可能です．

なお，ツールのダウンロードにはユーザー登録が必要です．

■ **発売元**：**ザイリンクス（株）**
■ **URL**：http://www.xilinx.co.jp/webpack/index.htm

第5回 続・C言語をコンパイルする際に指定するオプション

岸 哲夫

C言語をコンパイルする際に指定するオプションの説明を続ける.
今回は,
- ハードウェアモデルとコンフィグレーションオプション．補足として，クロスコンパイル環境について
- コード生成規約に対するオプション
- 実行に影響を与える環境変数
- プログラムにプロトタイプを追加するprotoizeについて

などの事柄について，説明と検証を行う．（筆者）

クロスコンパイル環境について

「クロスコンパイル」というのは，簡単にいえば，ターゲットマシンで実行可能なバイナリをホストマシンで作ることです．

GCCがターゲットマシンにできる環境については，前回説明したとおりです．

たとえば，日立のSH-4用のバイナリを作成するには，それ専用のライブラリをリンクしなければなりません．「ターゲット機種とコンパイラバージョンの指定オプション」で行った説明を補足すると，`-b`オプションの指定の際に`-b sh4`と指定したときと，指定しないときでリンクするライブラリを変えたり，実行するプリプロセッサ，狭義のコンパイラ，アセンブラ，リンカを変えることが可能です．ただし，通常は間違いを避けるために，クロスコンパイル用GCCは別の名称にして別のディレクトリに置くのが普通です．

例として，日立のSH-4用クロスコンパイル開発環境を構築してみましょう．

まずbinutil，GCC，newlibをダウンロードします．

GCCはバージョン2の最終版である`GCC-2.95.3.tar.gz`を`http://gcc.gnu.org/`からダウンロードします．

binutilは最新版の`binutils-2.13.tar.gz`を，`http://www.dnsbalance.ring.gr.jp/`からダウンロードします．

newlibは最新版の`newlib-1.10.0.tar.gz`を`http://sources.redhat.com/newlib/`からダウンロードします．

binutilはアセンブラ・リンカその他のツール類を含んでいます．newlibは標準Cライブラリです．RedHatのWebページにありますが，ほかのディストリビューションでも問題ありません．

まずはbinutilsの作成です．

```
$ tar zxvf binutils-2.13.tar.gz
$ cd binutils-2.13
$ ./configure --target=sh-hitachi-coff
                              --prefix=/usr/local
$ make
$ make install
```

以上でbinutilsが作成されました．エラーが起きる場合は環境をチェックしてください．

次にGCCの作成です．newlibのヘッダを使うので，両方とも解凍します．

```
$ tar zxvf gcc-2.95.3.tar.gz
$ tar zxvf newlib-1.10.0.tar.gz
$ cd gcc-2.95.3
$ mkdir work
$ cd work
$ ../configure --target=sh-hitachi-coff
      --prefix=/usr/local --with-newlib
      --with-headers=/newlib-1.10.0/
                         newlib/libc/include
$ make LANGUAGES="c"
$ make install LANGUAGES="c"
```

これでクロスコンパイル用GCCが作成されました．エラーが起きた場合，やはり環境をチェックしてください．

次にnewlibの作成です．

```
$ cd newlib-1.10.0
$ mkdir work
$ cd work
$ ../configure --target=sh-hitachi-coff
                         --prefix=/usr/local
$ make
$ make install
```

これでクロスコンパイル環境が作成されました．

```
main()
{
        printf("hello world!!¥n");
}
```

このように単純な"hello world"を`testsh.c`として保存します．

```
$ sh-hitachi-coff-gcc testsh.c
```

このようにすると`a.out`が作成されます．これはSHのcoff

〔リスト 1〕SH用にコンパイルしたもの(testsh.s)

```
        .file   "test.c"
        .data
gcc2_compiled.:
___gnu_compiled_c:
        .text
        .align 2
LC0:
        .ascii "hello world!!¥12¥0"
        .align 2
        .global _main
_main:
        mov.l   r8,@-r15
        mov.l   r14,@-r15
        sts.l   pr,@-r15
        mov     r15,r14
        mov.l   L3,r8
        jsr     @r8
        nop
        mov     r15,r1
        mov.l   L4,r2
        mov     r2,r4
        mov.l   L5,r8
        jsr     @r8
        nop
L2:
        mov     r14,r15
        lds.l   @r15+,pr
        mov.l   @r15+,r14
        mov.l   @r15+,r8
        rts
        nop
L6:
        .align 2
L3:
        .long   ___main
L4:
        .long   LC0
L5:
        .long   _printf
```

環境で動作します.

では，アセンブラコードで比較してみましょう(**リスト 1**，**リスト 2**).

```
$ cp testsh.c test.c
$ sh-hitachi-coff-gcc -S testsh.c
$ gcc -S test.c
```

このように，testsh.sのコードはインテルアーキテクチャとは異質のものです.

クロスコンパイル環境を構築すれば，SigmarionやLinuxザウルスSL-A300のためのバイナリを作成することも可能です.

〔リスト 2〕通常のGCCによるコンパイル(test.s)

```
        .file   "test.c"
        .version        "01.01"
gcc2_compiled.:
                .section        .rodata
.LC0:
        .string "hello world!!¥n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.96 20000731
                                (Red Hat Linux 7.3 2.96-110)"
```

〔リスト 3〕デフォルトでコンパイルしたアセンブラソース(test31.s)

```
        .file   "test31.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d¥n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        addl $-8,%esp
        call test1
        movl %eax,%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $16,%esp
        addl $-8,%esp
        call test2
        movl %eax,%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $16,%esp
        xorl %eax,%eax
        jmp .L2
        .p2align 4,,7
.L2:
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .align 4
.globl test1
        .type    test1,@function
test1:
        pushl %ebp
        movl %esp,%ebp
        movl $100,%eax
        jmp .L3
        .p2align 4,,7
.L3:
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe2:
        .size    test1,.Lfe2-test1
        .align 4
.globl test2
        .type    test2,@function
test2:
        pushl %ebp
        movl %esp,%ebp
        movl $200,%eax
        jmp .L4
        .p2align 4,,7
.L4:
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe3:
        .size    test2,.Lfe3-test2
        .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

ちなみに，Sigmarion は CPU に SH-4，SL-A300 は XScale をそれぞれ使用しているので，コンフィグレーションを適切にセットすれば問題なく動作するでしょう．

クロスコンパイルの詳細については，GCC のインストールの回で詳しく説明します．

コード生成規約に対するオプション

以下のオプションは，インターフェース規約を制御するものです．これはコード生成処理において使われます．

なお，ほとんどのオプションには，肯定形式と否定形式の両方があります．-fhoge の否定形式は-fno-hoge となります．

● -fexceptions

C++ の例外処理を有効にします．詳しくは C++ の回で説明します．通常，例外処理を必要としない C 言語では，デフォルトでこのオプションは無効となります．ただし，C++ で書かれた例外処理とリンクしている場合は，このオプションを有効にする必要があります(**リスト 3～リスト 5**)．

このように-fexceptions オプションを付けると，すべての関数に対してフレーム解放のための情報が生成されます．ライブラリ関数である printf も例外ではありません．

● -fpcc-struct-return

すべての struct 型や union 型の値を，レジスタに入れるのではなくメモリ上に置いて返します．

〔**リスト 4**〕-fexceptions **オプションを付けてコンパイルしたアセンブラソース**(test32.s)

```
        .file   "test32.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d¥n"
.text
        .align 4
.globl main
        .type    main,@function
main:
.LFB1:
        pushl %ebp
.LCFI0:
        movl %esp,%ebp
.LCFI1:
        subl $8,%esp
.LCFI2:
        addl $-8,%esp
        call test1
        movl %eax,%eax
        pushl %eax
        pushl $.LC0
.LCFI3:
        call printf
        addl $16,%esp
        addl $-8,%esp
.LCFI4:
        call test2
        movl %eax,%eax
        pushl %eax
        pushl $.LC0
.LCFI5:
        call printf
        addl $16,%esp
        xorl %eax,%eax
        jmp .L2
        .p2align 4,,7
.L2:
        movl %ebp,%esp
        popl %ebp
        ret
.LFE1:
.Lfe1:
        .size    main,.Lfe1-main
        .align 4
.globl test1
        .type    test1,@function
test1:
.LFB2:
        pushl %ebp
.LCFI6:
        movl %esp,%ebp
.LCFI7:
        movl $100,%eax
        jmp .L3
        .p2align 4,,7
.L3:
        movl %ebp,%esp
        popl %ebp
        ret
.LFE2:
.Lfe2:
        .size    test1,.Lfe2-test1
        .align 4
.globl test2
        .type    test2,@function
test2:
.LFB3:
        pushl %ebp
.LCFI8:
        movl %esp,%ebp
.LCFI9:
        movl $200,%eax
        jmp .L4
        .p2align 4,,7
.L4:
        movl %ebp,%esp
        popl %ebp
        ret
.LFE3:
.Lfe3:
        .size    test2,.Lfe3-test2

.section        .eh frame,"aw",@progbits
_ FRAME BEGIN   :
        .4byte  .LLCIE1
.LSCIE1:
        .4byte  0x0
        .byte   0x1
        .byte   0x0
        .byte   0x1
        .byte   0x7c
        .byte   0x8
        .byte   0xc
        .byte   0x4
        .byte   0x4
        .byte   0x88
        .byte   0x1
        .align 4
.LECIE1:
        .set    .LLCIE1,.LECIE1-.LSCIE1
        .4byte  .LLFDE1
.LSFDE1:
        .4byte  .LSFDE1-  FRAME BEGIN
        .4byte  .LFB1
        .4byte  .LFE1-.LFB1
        .byte   0x4
        .4byte  .LCFI0-.LFB1
        .byte   0xe
        .byte   0x8
        .byte   0x85
        .byte   0x2
        .byte   0x4
        .4byte  .LCFI1-.LCFI0
        .byte   0xd
        .byte   0x5
        .byte   0x4
        .4byte  .LCFI3-.LCFI1
        .byte   0x2e
        .byte   0x10
        .byte   0x4
        .4byte  .LCFI4-.LCFI3
        .byte   0x2e
        .byte   0x0
        .byte   0x4
        .4byte  .LCFI5-.LCFI4
        .byte   0x2e
        .byte   0x10
        .align 4
.LEFDE1:
        .set    .LLFDE1,.LEFDE1-.LSFDE1
        .4byte  .LLFDE3
.LSFDE3:
        .4byte  .LSFDE3-  FRAME BEGIN
        .4byte  .LFB2
        .4byte  .LFE2-.LFB2
        .byte   0x4
        .4byte  .LCFI6-.LFB2
        .byte   0xe
        .byte   0x8
        .byte   0x85
        .byte   0x2
        .byte   0x4
        .4byte  .LCFI7-.LCFI6
        .byte   0xd
        .byte   0x5
        .align 4
.LEFDE3:
        .set    .LLFDE3,.LEFDE3-.LSFDE3
        .4byte  .LLFDE5
.LSFDE5:
        .4byte  .LSFDE5-  FRAME BEGIN
        .4byte  .LFB3
        .4byte  .LFE3-.LFB3
        .byte   0x4
        .4byte  .LCFI8-.LFB3
        .byte   0xe
        .byte   0x8
        .byte   0x85
        .byte   0x2
        .byte   0x4
        .4byte  .LCFI9-.LCFI8
        .byte   0xd
        .byte   0x5
        .align 4
.LEFDE5:
        .set    .LLFDE5,.LEFDE5-.LSFDE5
        .ident  "GCC: (GNU) 2.95.3
                        20010315 (release)"
```

〔リスト5〕**Cソース**(test31.c)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char* argv[])
{
    printf("%d¥n",test1());
    printf("%d¥n",test2());
    return  0;
}
test1()
{
    return  100;
}
test2()
{
    return  200;
}
```

メモリ効率に問題がありますが，このコードの利点は，GCCによってコンパイルされたファイルと，他のコンパイラでコンパイルされたファイルの間で相互に呼び出しが可能になるという点です．構造体をメモリ上に置いて返すための正確な規約は，ターゲットのコンフィグレーションマクロに依存します(**リスト6**～**リスト8**).

このようにtest33.s(**リスト6**)では関数test1から戻る際にレジスタebpにセットして戻りますが，test34.s(**リスト7**)ではstruct tmのサイズである4バイトのアドレスを戻しています．

もっとも，実際にこのような手法を使用するかどうか疑問です．可読性/可搬性を高めるためには使用しないほうが良いと思います．

● -freg-struct-return

struct型とunion型の値をレジスタに入れて戻す規約を使います．サイズの小さい構造体に関しては，こちらのほうが-fpcc-struct-returnよりも効率的です．-fpcc-struct-returnとその否定である-freg-struct-returnのどちらも指定しないと，GCCはこの二つの規約のうちターゲットにとって標準であるものをデフォルトとして使用します．

● -fshort-enums

enum型に対して必要となるだけのバイト数を割り当てます．

宣言のなかで示された値の最大値がintで収まる場合，そのenum型はintと等しくなります．

実際にはPC/AT互換機の規格でintが32ビットである現在は，この指定をしても意味を成しません．enum型がintであると規定されているので，それ以下の大きさにはならないのです．

enum型にintの範囲を超える値を指定した場合，enum型が8バイト長になることは可能です．その場合，プログラム中でenum型をint型であると規定してしまうとバグの元になるので注意してください．

● -fshort-double

本来はdouble型のサイズはfloat型より大きいのですが，このオプションを指定すると同一のサイズとなります(**リスト9**).

プログラムを実行してサイズを表示すると，以下のようになります．

```
$ gcc -fshort-double test35.c
$ ./a.out
doubleのサイズは4です
floatのサイズは4です
$ gcc test35.c
$ ./a.out
doubleのサイズは8です
floatのサイズは4です
$
```

〔リスト6〕-fno-pcc-struct-return**オプションでコンパイルしたアセンブラソース**(test33.s)

```
        .file   "test33.c"
        .version        "01.01"
gcc2 compiled.:
.section        .rodata
.LC0:
        .string "%d¥n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp
        movl $60,-4(%ebp)
        addl $-12,%esp
        movl -4(%ebp),%eax
        pushl %eax
        call test1
        addl $16,%esp
        movl %eax,%eax
        movl %eax,-8(%ebp)
        addl $-8,%esp
        movl -4(%ebp),%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $16,%esp
        addl $-8,%esp
        movl -8(%ebp),%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $16,%esp
        xorl %eax,%eax
        jmp .L2
        .p2align 4,,7
.L2:
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .align 4
.globl test1
        .type    test1,@function
test1:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%edx
        movl %edx,%eax
        jmp .L3
        .p2align 4,,7
.L3:
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe2:
        .size    test1,.Lfe2-test1
        .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

〔リスト7〕-fpcc-struct-returnオプションを付けてコンパイルしたアセンブラソース(test34.s)

```
        .file   "test34.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "%d¥n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp
        movl $60,-4(%ebp)
        leal -8(%ebp),%eax
        addl $-8,%esp
        movl -4(%ebp),%edx
        pushl %edx
        pushl %eax
        call test1
        addl $12,%esp
        addl $-8,%esp
        movl -4(%ebp),%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $16,%esp
        addl $-8,%esp
        movl -8(%ebp),%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $16,%esp
        xorl %eax,%eax
        jmp .L2
        .p2align 4,,7
.L2:
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .align 4
.globl test1
        .type    test1,@function
test1:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax
        movl 12(%ebp),%edx
        movl %edx,(%eax)
        jmp .L3
.L3:
        movl %eax,%eax
        movl %ebp,%esp
        popl %ebp
        ret $4
.Lfe2:
        .size    test1,.Lfe2-test1
        .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

〔リスト8〕Cソース(test33.c)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct tm
{
  int tm_sec;
};
struct tm test1(struct tm hoge);
int main(int argc, char* argv[])
{
    struct tm testtm;
    struct tm testtm1;
    testtm.tm_sec   =60;
    testtm1 =   test1(testtm);
    printf("%d¥n",testtm.tm_sec);
    printf("%d¥n",testtm1.tm_sec);
    return  0;
}
struct tm test1(struct tm hoge)
{
    return  hoge;
}
```

〔リスト9〕サイズを表示するCソース(test35.c)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char* argv[])
{
    double  d_wk;
    float   f_wk;
    printf("doubleのサイズは%dです¥n",sizeof(d_wk));
    printf("floatのサイズは%dです¥n",sizeof(f_wk));
    return  0;
}
```

〔リスト10〕グローバル変数が重複しているCソース(test36.c)

```
#include <stdio.h>
void    test1();
void    test2();
int     ix;
int main(int argc, char* argv[])
{
    test1();
    test2();
    return  0;
}
void    test1()
{
    printf("test1¥n");
}
```

● -fshared-data

このオプションを指定してコンパイルした範囲では，データおよびconst指定されていない変数は，プライベートデータではなく共有データとするよう要求します．

一見使いやすいように思えますが，バグを誘発する危険度が大きいと思います．このような場合，スレッドを利用してmutexでデータを管理する方法を選択したほうが安全です．通常のGNU/Linuxではこのような共有データの使い方はできません．

● -fno-common

初期化済みでないグローバル変数をオブジェクトファイル中のbssセクションに割り当てます．

二つの異なるコンパイル単位の中でexternを使わずに同一の変数が宣言されていると，リンクする際にエラーが発生するようになるという効果があります(**リスト10**)．

```
$ gcc -fcommon -o test36 test36.c test37.c
$ gcc -fno-common -o test36 test36.c test37.c
/tmp/ccojxEZU.o(.bss+0x0):
                    multiple definition of `ix'
/tmp/cckUp9bG.o(.bss+0x0):
                                first defined here
collect2: ld returned 1 exit status
$
```

● -fno-ident

このオプションは#ident指示子を無視します．

#ident指示子は機種によって有効に使用することが可能ですが，汎用的ではない話なので省略します．

● -fno-gnu-linker

GNU以外のリンカを使う場合に指定しますが，C++の場合はコンストラクタやデコンストラクタのリンクができなくなる恐れがあります．そのような場合には，collect2というソフトウェアを使用します．

● -finhibit-size-directive

一般的なプログラムのコンパイルには関係のない話ですが，このオプションはアセンブラの.size指示子を出力しません．関数が途中で分割されて主記憶上にロードされると問題を引き起こすことになるものは分割しません．

このオプションは，特殊なプログラムソースcrtstuff.cをコンパイルする際に使われます．これ以外では使用しません．

● -fverbose-asm

生成されるアセンブラコードをより読みやすくするために，そのアセンブラコードの中に余分なコメント情報を追加します．このオプションはコンパイラ自身をデバッグする際にしか役に立ちません．デフォルトは-fno-verbose-asmです．

● -fvolatile

ポインタを経由したメモリ参照は，すべてvolatile指定されたものとみなします．

volatile宣言されていない自動変数は，破壊される可能性があります．また最適化などで意図しないアドレスを参照するコードにされてしまった場合，それがハードウェアの入出力に関わるとき困ったことになります．

そのような状況を防止するために，このオプションが役立ちます．ただし，volatile指定すべきものは意図的にしたほうが，可読性も可搬性も高まると思います．

● -fvolatile-global

あらゆるデータ項目に対するメモリ参照はすべて暗黙にvolatile指定されたものとします．しかしstaticデータ項目をvolatile指定されたものとはみなしません．

これもやはり明示的にvolatile指定すべきだと思います．

● -fvolatile-static

staticデータに対するメモリ参照は，すべて暗黙にvolatile指定されたものとします．

● -fpic

ターゲットマシンにおいてサポートされていれば，共用ライブラリにおいて使用するのに適したposition-independent cコードを生成します．

このコードは，すべての固定アドレスにグローバルオフセットテーブルを通じてアクセスします．つまり再配置可能なコードになります．プログラムが起動するときに，ダイナミックローダがグローバルオフセットテーブルのエントリを解決します．

一般的には，ELF実行ファイルを作成する際に使用されるテクニックです．

● -fPIC

上の-fpicを指定してもマシンによってはグローバルオフセットテーブルの上限サイズが決められています．

その場合，このオプションを指定すれば回避可能です．

● -ffixed-reg

regで指定される名前のレジスタを，生成されたコード中で自動的に割り当てません(**リスト12～リスト14**)．

この例では以下のようなコンパイルを行いました．

```
$ gcc -S -ffixed-si -ffixed-di -ffixed-ax
                          -ffixed-bx  test38.c
```

このようなオプションをつけた場合，si，di，ax，bxの各レジスタに値を割り当てません．

ソース中のインラインアセンブラではbxを明示的に使っているので，bxはその命令文だけで使用されています．

リスト14では，

```
movl $10,%ebx
movl $20,%esi
movl $30,%edi
```

のように値が割り当てられていますが，**リスト13**では，

```
movl $10,-4(%ebp)
movl $20,-8(%ebp)
movl $30,-12(%ebp)
```

のようにメモリ上に割り当てられているのがわかります．

用途としては，インラインアセンブラ中であるレジスタを独立させて使用したいときなどがあります．

● -fcall-used-reg

regで指定される名前のレジスタを，関数呼び出しで内容が破壊されてもよいレジスタとして取り扱います．

このオプションを指定してコンパイルされた関数は，指定したレジスタの待避，復元を行いません．固定的な役割をもつレジスタを指定してこのフラグを使うと，正常に動作しない恐れがあります．

● -fcall-saved-reg

regで指定される名前のレジスタを，関数呼び出しにより内容が破壊される前に退避し，呼び出し後に復元されるレジスタとして取り扱います．

やはり固定的な役割をもつレジスタを指定してこのフラグを使うと正常に動作しない恐れがあります．

〔**リスト11**〕**グローバル変数が重複しているCソース**(test37.c)

```
#include <stdio.h>
int      ix;
void     test2()
{
    printf("test2¥n");
}
```

〔リスト12〕-ffixed-reg 検証用 C ソース(test38.c)

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    register        a;
    register        b;
    register        c;
    register        d;
    register        e;
    register        f;
    register        g;
    a   =   10;
    b   =   20;
    c   =   30;
    d   =   40;
    e   =   50;
    f   =   60;
    g   =   70;
    printf("test%d,%d,%d,%d,%d,%d,%d¥n",a,b,c,d,e,f,g);
    asm ("mov %ebx,$1000 ");
    return  0;
}
```

〔リスト13〕-ffixed-reg オプションでコンパイルしたもの(test38.s)

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    register        a;
    register        b;
    register        c;
    register        d;
    register        e;
    register        f;
    register        g;
    a   =   10;
    b   =   20;
    c   =   30;
    d   =   40;
    e   =   50;
    f   =   60;
    g   =   70;
    printf("test%d,%d,%d,%d,%d,%d,%d¥n",a,b,c,d,e,f,g);
    asm ("mov %ebx,$1000 ");
    return  0;
}
```

〔リスト14〕オプションなしでコンパイルしたもの(オプションなしtest38.s)

```
        .file   "test38.c"
        .version        "01.01"
gcc2 compiled.:
.section        .rodata
.LC0:
        .string "test%d,%d,%d,%d,%d,%d,%d¥n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $28,%esp
        pushl %edi
        pushl %esi
        pushl %ebx
        movl $10,%ebx
        movl $20,%esi
        movl $30,%edi
        movl $40,-4(%ebp)
        movl $50,-8(%ebp)
        movl $60,-12(%ebp)
        movl $70,-16(%ebp)
        movl -16(%ebp),%eax
        pushl %eax
        movl -12(%ebp),%eax
        pushl %eax
        movl -8(%ebp),%eax
        pushl %eax
        movl -4(%ebp),%eax
        pushl %eax
        pushl %edi
        pushl %esi
        pushl %ebx
        pushl $.LC0
        call printf
        addl $32,%esp
#APP
        mov %ebx,$1000
#NO APP
        xorl %eax,%eax
        jmp .L2
.L2:
        leal -40(%ebp),%esp
        popl %ebx
        popl %esi
        popl %edi
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

● -fpack-struct

すべての構造体メンバの間を空けることなく詰めます．このオプションを使うと，生成されるコードは最適なものではなくなります．

そして，構造体メンバのオフセットはライブラリ関数の呼び出しフォーマットと一致しなくなります(**リスト15～リスト17**，次頁)．

詰めてコンパイルしたものは，構造体の並びが以下のものに対し，

```
struct tm_wk
{
  int tm_sec;
  int tm_min;
  int tm_hour;
  char     x;
  int tm_mday;
  int tm_mon;
  int tm_year;
  double   y;
};
```

メモリ配置が，

```
movl $10,-36(%ebp)
movl $20,-32(%ebp)
movl $30,-28(%ebp)
movb $97,-24(%ebp)
movl $40,-23(%ebp)
movl $50,-19(%ebp)
movl $60,-15(%ebp)
```

のように構造体のバイト数そのままですが，通常は，

```
movl $10,-36(%ebp)
```

〔リスト 15〕-fpack-struct 検証用 C ソース(test39.c)

```
#include <time.h>
#include <sys/time.h>
#include <stdio.h>
struct tm_wk
{
  int tm_sec;           /* Seconds. [0-60] (1 leap second) */
  int tm_min;           /* Minutes. [0-59] */
  int tm_hour;          /* Hours.   [0-23] */
  char  x;
  int tm_mday;          /* Day.     [1-31] */
  int tm_mon;           /* Month.   [0-11] */
  int tm_year;          /* Year - 1900.  */
  double    y;
};
struct tm_wk    test(struct tm_wk intm);
int main(int argc, char* argv[])
{
    struct tm_wk    wk;
    struct tm_wk    outwk;
    wk.tm_sec    =   10;
    wk.tm_min    =   20;
    wk.tm_hour   =   30;
    wk.x     =   'a';
    wk.tm_mday   =   40;
    wk.tm_mon    =   50;
    wk.tm_year   =   60;
    wk.y     =   0;
    outwk    =   test(wk);
    return  0;
}
struct tm_wk    test(struct tm_wk intm)
{
    struct tm_wk    wk;
    wk.tm_sec    =   50;
    return  wk;
}
```

〔リスト 16〕構造体メンバの間を詰めてコンパイルしたもの(test39_pack.s)

```
        .file   "test39.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $80,%esp
        pushl %edi
        pushl %esi
        movl $10,-36(%ebp)
        movl $20,-32(%ebp)
        movl $30,-28(%ebp)
        movb $97,-24(%ebp)
        movl $40,-23(%ebp)
        movl $50,-19(%ebp)
        movl $60,-15(%ebp)
        movl $0,-11(%ebp)
        movl $0,-7(%ebp)
        leal -72(%ebp),%eax
        addl $-8,%esp
        addl $-36,%esp
        movl %esp,%edi
        leal -36(%ebp),%esi
        cld
        movl $8,%ecx
        rep
        movsl
        movsb
        pushl %eax
        call test
        addl $44,%esp
        xorl %eax,%eax
        jmp .L2
        .p2align 4,,7
.L2:
        leal -88(%ebp),%esp
        popl %esi
        popl %edi
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .align 4
.globl test
        .type    test,@function
test:
        pushl %ebp
        movl %esp,%ebp
        subl $48,%esp
        pushl %edi
        pushl %esi
        movl 8(%ebp),%eax
        movl $50,-36(%ebp)
        movl %eax,%edi
        leal -36(%ebp),%esi
        cld
        movl $8,%ecx
        rep
        movsl
        movsb
        jmp .L3
.L3:
        movl %eax,%eax
        leal -56(%ebp),%esp
        popl %esi
        popl %edi
        movl %ebp,%esp
        popl %ebp
        ret $4
.Lfe2:
        .size    test,.Lfe2-test
        .ident  "GCC: (GNU) 2.95.3
                        20010315 (release)"
```

〔リスト 17〕通常のコンパイルをしたもの(test39_unpack.s)

```
        .file   "test39.c"
        .version        "01.01"
gcc2 compiled.:
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $80,%esp
        pushl %edi
        pushl %esi
        movl $10,-36(%ebp)
        movl $20,-32(%ebp)
        movl $30,-28(%ebp)
        movb $97,-24(%ebp)
        movl $40,-20(%ebp)
        movl $50,-16(%ebp)
        movl $60,-12(%ebp)
        movl $0,-8(%ebp)
        movl $0,-4(%ebp)
        leal -72(%ebp),%eax
        addl $-8,%esp
        addl $-36,%esp
        movl %esp,%edi
        leal -36(%ebp),%esi
        cld
        movl $9,%ecx
        rep
        movsl
        pushl %eax
        call test
        addl $44,%esp
        xorl %eax,%eax
        jmp .L2
        .p2align 4,,7
.L2:
        leal -88(%ebp),%esp
        popl %esi
        popl %edi
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .align 4
.globl test
        .type    test,@function
test:
        pushl %ebp
        movl %esp,%ebp
        subl $48,%esp
        pushl %edi
        pushl %esi
        movl 8(%ebp),%eax
        movl $50,-36(%ebp)
        movl %eax,%edi
        leal -36(%ebp),%esi
        cld
        movl $9,%ecx
        rep
        movsl
        jmp .L3
.L3:
        movl %eax,%eax
        leal -56(%ebp),%esp
        popl %esi
        popl %edi
        movl %ebp,%esp
        popl %ebp
        ret $4
.Lfe2:
        .size    test,.Lfe2-test
        .ident  "GCC: (GNU) 2.95.3
                        20010315 (release)"
```

```
    movl $20,-32(%ebp)
    movl $30,-28(%ebp)
    movb $97,-24(%ebp)
    movl $40,-20(%ebp)
    movl $50,-16(%ebp)
    movl $60,-12(%ebp)
```

のように，4バイト境界で領域が取られます．

● -fcheck-memory-usage

このオプションは個々のメモリアクセスをチェックするための追加のコードを生成します．Checkerという不正なメモリアクセスを検出するツールがあり，このオプションで作成されたコードはそのツールに対応しています．

なお，このオプションを指定すると，メモリチェックが有効となっている関数の中では，asmキーワードや__asm__キーワードは使えません．

Checkerを使用しなければ関係ないオプションです．**リスト18～リスト20**を参照するとわかりますが，生成されたコード中ではchkr_check_addrおよびchkr_set_rightへの呼び出しを行っています．

● -fprefix-function-name

このオプションは，関数名に対して生成されるシンボルに接頭語を付加します．GCCは，呼び出される関数だけでなく，定義された関数の名前にも接頭語を付加します．このオプションは上のオプションと対にして使用します．

● -finstrument-functions

このオプションは，関数の入口と出口にプロファイル用の呼び出しを生成します．

〔リスト18〕-fcheck-memory-usage検証用Cソース(test40.c)

```
test()
{
    char    a;
    a='a';
}
```

〔リスト19〕メモリアクセスをチェックするための追加のコードを付加してコンパイルしたもの(test40a.s)

```
        .file   "test40.c"
        .version        "01.01"
gcc2 compiled.:
.globl chkr_set_right
.globl chkr_check_addr
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $20,%esp
        pushl %ebx
        addl $-4,%esp
        pushl $3
        movl %esp,%eax
        addl $-4,%esp
        pushl $3
        pushl $4
        pushl %eax
        call chkr_set_right
        addl $16,%esp
        pushl $4
        movl %esp,%eax
        addl $-4,%esp
        pushl $3
        pushl $4
        pushl %eax
        call chkr_set_right
        addl $16,%esp
        leal 8(%ebp),%eax
        pushl %eax
        movl %esp,%eax
        addl $-4,%esp
        pushl $3
        pushl $4
        pushl %eax
        call chkr_set_right
        addl $16,%esp
        call chkr_set_right
        addl $16,%esp
        leal 12(%ebp),%ebx
        addl $-4,%esp
        pushl $3
        movl %esp,%eax
        addl $-4,%esp
        pushl $3
        pushl $4
        pushl %eax
        call chkr_set_right
        addl $16,%esp
        pushl $4
        movl %esp,%eax
        addl $-4,%esp
        pushl $3
        pushl $4
        pushl %eax
        call chkr_set_right
        addl $16,%esp
        pushl %ebx
        movl %esp,%eax
        addl $-4,%esp
        pushl $3
        pushl $4
        pushl %eax
        call chkr_set_right
        addl $16,%esp
        call chkr_set_right
        addl $16,%esp
        leal -4(%ebp),%ebx
        addl $-4,%esp
        pushl $2
        movl %esp,%eax
        addl $-4,%esp
        pushl $3
        pushl $4
        pushl %eax
        call chkr_set_right
        addl $16,%esp
        pushl $4
        movl %esp,%eax
        addl $-4,%esp
        pushl $3
        pushl $4
        pushl %eax
        call chkr_set_right
        addl $16,%esp
        pushl %ebx
        movl %esp,%eax
        addl $-4,%esp
        pushl $3
        pushl $4
        pushl %eax
        call chkr_set_right
        addl $16,%esp
        call chkr_check_addr
        addl $16,%esp
        movl $30,-4(%ebp)
        jmp .L2
        .p2align 4,,7
.L2:
        movl -24(%ebp),%ebx
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.95.3
                        20010315 (release)"
```

〔リスト20〕通常のコンパイルをしたもの(test40.s)

```
        .file   "test40.c"
        .version        "01.01"
gcc2 compiled.:
.text
        .align 4
.globl test
        .type    test,@function
test:
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp
        movb $97,-1(%ebp)
.L2:
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe1:
        .size    test,.Lfe1-test
        .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

関数の中に入った直後と関数から出る直前に，次の名称のプロファイル用の関数が呼び出されます．引き数は，対象の関数のアドレスとその呼び出し箇所です．

プロトタイプは次のようになります．

```
void __cyg_profile_func_enter
            (void *this_fn, void *call_site);
void __cyg_profile_func_exit
            (void *this_fn, void *call_site);
```

前出のソース`test40.c`（**リスト18**）でこのオプションを付けて生成されたコードは，**リスト21**のとおりです．

- **-fstack-check**

スタックの境界を超えないようにチェックするコードを生成します．マルチスレッド環境ではこのフラグを指定すべきです．

シングルスレッド環境下では，ほとんどすべてのシステムにおいてスタックオーバフローは自動的に検出されるので，このオプションを指定する意味はありません．

- **-fargument-alias**
- **-fargument-noalias**
- **-fargument-noalias-global**

このオプションをユーザーが自分で使う必要はありません．

このオプションは仮引き数間，および仮引き数とグローバルデータの間の可能な関連付けを指定します．

`-fargument-alias`は，引き数（仮引き数）がお互いに別名になっている可能性があること，それにグローバルデータの別名になっている可能性があることを指定します．

`-fargument-noalias`は，引き数はお互いに別名になっていることはないのですが，グローバルデータの別名になっている可能性があることを指定します．

`-fargument-noaliase-global`は，引き数はお互いに別名になっていないし，グローバルデータの別名にもなっていないことを指定します．

- **-fleading-underscore**

このオプションとその否定のオプションである`-fno-leading-underscore`は，オブジェクトファイルの中でCのシンボルが表現される方法を強制的に変更します．古いアセンブリコードとのリンクをサポートします．

このオプションの指定が何をもたらすかを理解した上で使用しないと大きな混乱を招きますので，注意して使ってください．

なお，すべてのターゲットにおいて完全にサポートされているわけではありません．

前出のソース`test40.c`（**リスト18**）でこのオプションを付けて生成されたコードは，**リスト22**のとおりです．

このように先頭に__（アンダースコア）が付いています．

GCCのオプションの説明はこれで終わります．次に，環境変数について説明と検証を行います．

実行に影響を与える環境変数

ここでは，GCCの実行時に影響を及ぼす環境変数について記します．

ファイルを探索する際に利用されるディレクトリ，または接頭語を指定することによって作用を及ぼします．また，環境変数はコンパイル環境の他の側面を指定するためにも使われます．探索される場所については，`-B`，`-I`，`-L`のようなオプションを使うことによっても指定可能であることに注意してください（第4回で説明した「ディレクトリ探索のためのオプション」を参照）．

〔リスト21〕プロファイル用の呼び出しを付加したアセンブラソース（test41.s）

```
        .file   "test41.c"
        .version        "01.01"
gcc2 compiled.:
.globl __cyg_profile_func_enter
.globl __cyg_profile_func_exit
.text
        .align 4
.globl test
        .type    test,@function
test:
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp
        addl $-8,%esp
        movl 4(%ebp),%eax
        pushl %eax
        pushl $test
        call __cyg_profile_func_enter
        addl $16,%esp
        movb $97,-1(%ebp)
.L2:
        addl $-8,%esp
        movl 4(%ebp),%eax
        pushl %eax
        pushl $test
        call __cyg_profile_func_exit
        addl $16,%esp
        movl %eax,%eax
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe1:
        .size    test,.Lfe1-test
        .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

〔リスト22〕Cのシンボルが表現される方法を強制的に変更したアセンブラソース（test42.s）

```
        .file   "test42.c"
        .version        "01.01"
gcc2 compiled.:
.text
        .align 4
.globl _test
        .type    _test,@function
_test:
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp
        movb $97,-1(%ebp)
.L2:
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe1:
        .size    _test,.Lfe1-_test
        .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

もちろん，コマンドラインオプションによる指定は，環境変数による指定よりも優先されます．一方，環境変数による指定は，GCCのコンフィグレーションにおける指定よりも優先されます．

- LANG
- LC_CTYPE
- LC_MESSAGES
- LC_ALL

以上の環境変数は，異なる国の慣習をサポートできるようにGCCがローカライズ情報を使う方法をコントロールします．

GCCは，configureによってそのようにするように構成されている場合には，ロケールカテゴリLC_CTYPE，LC_MESSAGESを調べます．これらのロケールカテゴリには，インストール環境によりサポートされている任意の値をセットすることができます．

日本語のEUC漢字コードを使う環境ならば，LANGはja_JP.eucJPと，シフトJISならばja_JP.SJISとなっています．

環境変数LC_CTYPEは，文字分類を指定します．

環境変数LC_MESSAGESは，診断メッセージにおいて使用する言語を指定します．

環境変数LC_ALLがセットされると，その値によってLC_CTYPEやLC_MESSAGESの元の設定は無効にされます．環境変数LC_ALLがセットされていない場合は，LC_CTYPEとLC_MESSAGESのデフォルトの値は環境変数LANGの値となります．

これらの変数がいずれもセットされていない場合，GCCのデフォルトは英語環境となります．

環境変数だけ設定しても診断メッセージが日本語で出力されるわけではありません．それなりの環境構築が必要となります．

なお「Linuxにおける日本語ロケールに関する指針」という文書があります．どのような機能を提供すべきかの指針や，どのように実装すべきかの指針を提言しています．

- 「Linuxにおける日本語ロケールに関する指針」のWebページ
  http://www.linux.or.jp/JF/JFdocs/Japanese-Locale-Policy/index.html

- TMPDIR

一時ファイルを作成するのに使われるディレクトリを指定する際に使用します．

コンパイルの過程は，四つの段階に分けることができます．プリプロセス/コンパイル/アセンブル/リンクです．

GCCは，それぞれの段階の中間出力を保存するために一時ファイルを作成し，それが次の段階の入力として使われます．たとえば，コンパイルの出力はアセンブラソースです．オプション指定でアセンブラソースを意図的に作成しないかぎり，中間出力として一時ファイルに保存され，次の段階の入力として使われます．

- GCC_EXEC_PREFIX

GCC_EXEC_PREFIXがセットされていると，それはコンパイラにより実行される下位プログラムの名前の接頭語となります．

GCC_EXEC_PREFIXのデフォルトの値はprefix/lib/gcc-lib/です．prefixは，configureスクリプトを実行したときのprefixの値です．クロスコンパイル環境を構築した際にprefixの値は変わりますが，通常は/usrです．もちろん-Bオプションで指定された別の接頭語があれば，そちらが優先されます．

デフォルトの/usr/lib/gcc-lib/の下にはスタートアップルーチンや共用ライブラリなどがあります．

通常はありえませんが，クロスコンパイル環境においてスタートアップルーチンがリンクエラーになった場合，この環境変数が正しいかチェックしましょう．

- COMPILER_PATH

COMPILER_PATHの値は，PATHと同じくコロンで区切られたディレクトリのリストです．上で説明したGCC_EXEC_PREFIXを使ってcc1コマンドなどを見つけることができない場合，この環境変数で指定されたディレクトリを探索します．

- LIBRARY_PATH

LIBRARY_PATHの値は，PATHと同じくコロンで区切られたディレクトリのリストです．GCCがconfigureによってネイティブコンパイラとして構成された場合，GCC実行時に，GCC_EXEC_PREFIXを使って特殊なリンカファイルを見つけることができないと，この環境変数で指定されたディレクトリを探索します．

GCC実行時のリンク処理では，-lオプションで指定された通常のライブラリを探す際にも，このディレクトリが使われます．

もちろん，明示的にライブラリ探索用と宣言されたディレクトリである-Lオプションで指定されたものが最初に使われます．

- C_INCLUDE_PATH
- CPLUS_INCLUDE_PATH
- OBJC_INCLUDE_PATH

上の環境変数は特定の言語に関係するものです．個々の変数の値は，PATHと同じくコロンで区切られたディレクトリのリストです．GCC実行時にヘッダファイルを探す際には，まずオプション-Iで指定されたディレクトリが探索され，続いて上の環境変数のうち使用している言語に対応するものに設定されているディレクトリが探索されます．

標準のヘッダファイルディレクトリは，このあとに探索されます．

C_INCLUDE_PATHはC言語，CPLUS_INCLUDE_PATHはC++言語，OBJC_INCLUDE_PATHはOBJECTIVE C言語にそれぞれ対応します．

- DEPENDENCIES_OUTPUT

この変数がセットされていると，その値はコンパイラにより処理されるヘッダファイルに基づいてmake用の依存関係をどのように出力するかを指定します．この出力は，-Mオプションによる出力とよく似ていますが，ここでは別ファイルに書き込まれ，通常のコンパイル処理も行われます．

実際に指定してみましょう．次に示すようになります．

```
$ export DEPENDENCIES_OUTPUT=$HOME/Out.dat
$ gcc -M test39.c
test39.o: test39.c /usr/include/time.h ¥
 /usr/include/features.h ¥
 /usr/include/sys/cdefs.h ¥
 /usr/include/gnu/stubs.h ¥
 /usr/lib/gcc-lib/i586-pc-linux/2.95.3/
                                include/stddef.h¥
 /usr/include/bits/time.h ¥
 /usr/include/bits/types.h ¥
 /usr/include/bits/pthreadtypes.h ¥
 /usr/include/bits/sched.h ¥
 /usr/include/sys/time.h ¥
 /usr/include/sys/select.h ¥
 /usr/include/bits/select.h ¥
 /usr/include/bits/sigset.h ¥
 /usr/include/stdio.h ¥
 /usr/include/libio.h ¥
 /usr/include/_G_config.h ¥
 /usr/include/wchar.h ¥
 /usr/include/bits/wchar.h ¥
 /usr/include/gconv.h ¥
 /usr/lib/gcc-lib/i586-pc-linux/
                       2.95.3/include/stdarg.h¥
 /usr/include/bits/stdio_lim.h
$ gcc test39.c
$ cat Out.dat
test39.o: test39.c
$
```

● LANG

この環境変数は，GCCの実行時にロケール情報を渡すために使われます．この情報の用途の一つに文字セットの決定があります．C/C++において文字リテラル，文字列リテラル，コメントが解析される際に使われます．

GCCが構築時にconfigureによってマルチバイト文字を取り扱えるよう構成されている場合，LANGの値として以下のものが認識されます．

● C-JIS

JIS文字を認識します．

● C-SJIS

シフトJIS文字を認識します．

● C-EUCJP

EUC文字を認識します．

LANGが定義されていない場合や，値が不正な場合には，マルチバイト文字の認識と変換を行うために，デフォルトのロケールにより定義されているmblenとmbtowcを使うことになります．

プログラムにプロトタイプを追加するprotoizeについて，またプロトタイプを削除するunprotoizeについて

ツールであるprotoizeは，プログラムにプロトタイプを追加するために使用します．これにより，プログラムプロトタイプ宣言や引き数の型の取り扱いに関してANSI C方式に変換されます．一緒に提供されているunprotoizeがこの逆のことを行います．こちらは，プロトタイプを見つけると，そこから引き数の型情報を取り除きます．

これらのプログラムを実行する際には，ソースファイルをコマンドライン引き数として指定しなければなりません．変換プログラムは，ソースファイルの中でどのような関数が定義されているかを調べるために，まずそれらをコンパイルすることから始めます．

その後に変換を行いますが，カレントディレクトリにあるソースファイルやヘッダファイルだけを変換します．

あるディレクトリの下のファイルを変換したい場合には，-d directoryオプションでその追加のディレクトリを指定することが可能です．変換の対象外としたい特定のファイルを-x fileオプションで指定することも可能です．

protoizeによる基本的な変換は，引き数の型を指定するために関数定義や関数宣言を書き直すことです．可変個数の引き数を取る関数定義や関数宣言については変換しません．

protoizeは，ソース上で関数定義よりも前にある関数呼び出しから利用できるように，ソースファイルの先頭にプロトタイプ宣言を挿入するようにすることもできます．

また，宣言されていない関数が呼び出されているブロックの中に，ブロックスコープをもつプロトタイプ宣言を挿入することもできます．

unprotoizeによる基本的な変換では，ほとんどの関数宣言を書き直して引き数の型を取り除き，ANSI以前の旧方式の形式に関数定義を書き直します．

protoizeやunprotoizeからの出力は，元のソースファイルを置き換えます．元のファイルは，末尾が“.save”で終わる名前に変えられます．末尾が“.save”で終わる名前のファイルがすでに存在する場合は，ソースファイルは破棄されてしまいます．

以下に単純な変換の例を示します．

```
$ cat test42.c
test()
{
        char    a;
        a='a';
}
$
$ protoize test42.c
protoize: compiling `test42.c'
```

〔リスト 23〕プロトタイプ宣言のないソース(実行前 test43.c)

```
#include <stdio.h>
#include <pthread.h>
#include <sched.h>
typedef struct t_mutex
{
    int             flag;
    pthread_mutex_t mutex;
}  _mutex;
main()
{
    int res;
    _mutex  info;
    res =   thread_mutex_create(&info);
}
int thread_mutex_create(_mutex *mutex_info)
{
    int result;
    pthread_mutex_t buf;
    result  = pthread_mutex_init(&(mutex_info->mutex),0);
    if  (   result  ==  0   )
    {
        return  0;
    }
    else
    {
        return  -1;
    }
}
```

```
protoize: converting file `test42.c'
$ cat test42.c
test(void)
{
        char    a;
        a='a';
}
$
```

ANSIの規約では，引き数がない場合には“ void ”をつけることになっています．関数の引き数を変換した例です．

リスト 23，**リスト 24** の例は「プロトタイプ宣言」を挿入する例です．このように「プロトタイプ宣言」を挿入し，関数宣言の形式も変更してしまいます．ただし，現在の C の記法ではこういった方法をとらないように思えます．**リスト 25** のように「プロトタイプ宣言」を挿入し，関数宣言もこのようにしたほうがわかりやすいように思えます．

もっとも，この件に関してはプログラマの好みの問題です．通常は，protoize を使わないで最初からプロトタイプ宣言を挿入してコーディングしたほうがメンテナンス性が高まると思います．

*　　　　　*

次回は GCC のインストールに関して，また GCC の拡張機能について，説明と検証を詳細に行う予定です．

〔リスト 24〕protoize で処理したソース(実行後 test43.c)

```
#include <stdio.h>
#include <pthread.h>
#include <sched.h>
typedef struct t_mutex
{
    int             flag;
    pthread_mutex_t mutex;
}  _mutex;

int thread_mutex_create ();

main(void)
{
    int res;
    _mutex  info;
    res =   thread_mutex_create(&info);
}
int thread_mutex_create(mutex_info)
     _mutex *mutex_info;
{
    int result;
    pthread_mutex_t buf;
    result  = pthread_mutex_init(&(mutex_info->mutex),0);
    if  (   result  ==  0   )
    {
        return  0;
    }
    else
    {
        return  -1;
    }
}
```

〔リスト 25〕筆者の考えで修正したソース(test43.c)

```
#include <stdio.h>
#include <pthread.h>
#include <sched.h>
typedef struct t_mutex
{
    int             flag;
    pthread_mutex_t mutex;
}  _mutex;

int thread_mutex_create(_mutex *mutex_info);

main(void)
{
    int res;
    _mutex  info;
    res =   thread_mutex_create(&info);
}
int thread_mutex_create(_mutex *mutex_info)
{
    int result;
    pthread_mutex_t buf;
    result  = pthread_mutex_init(&(mutex_info->mutex),0);
    if  (   result  ==  0   )
    {
        return  0;
    }
    else
    {
        return  -1;
    }
}
```

きし・てつお　オフィス岸

組み込みプログラミングノウハウ入門

第8回 アクティブオブジェクトモデリングのはなし

藤倉俊幸

はじめに

UML2.0がいよいよ2003年にリリースされる．UMLの進化を**図1**にまとめる．UML2.0の目玉は，MDA(Model-Driven Architecture)をOMGが公式に表明したことであると，来日したBran Selic氏が語っていた．Bran Selic氏は組み込み用CASEツールのRose RealTimeの基礎となったROOM(Real-Time Object-Oriented Modeling)法[注1]の開発者で，UML2.0策定に貢献している．MDAとは，モデルをソフト開発の中心を成す成果物と考えることである．この考え方のもとでは，ソースファイルはコンパイラが生成するオブジェクトファイルと同じような位置付けになる．設計は実装自身になり，設計の実態はモデルを作ることである．そしてUMLは，モデルを記述するための言語である．

● モデリングとは

モデリングとはいうまでもなく「不必要な詳細を隠してより抽象的に」,「コンピュータの言葉でなくよりアプリケーションに近い言葉で」ソフトウェアを記述することである．われわれが現在「ソースコード」と呼んでいるものも，じつはアセンブラや機械語に変換されて実行される．この点を考えるといわゆる高級言語のソースコードも，レジスタなどの詳細を隠して，より人間に近い言葉でソフトウェアを記述したモデルである(**図2**).

● MDA

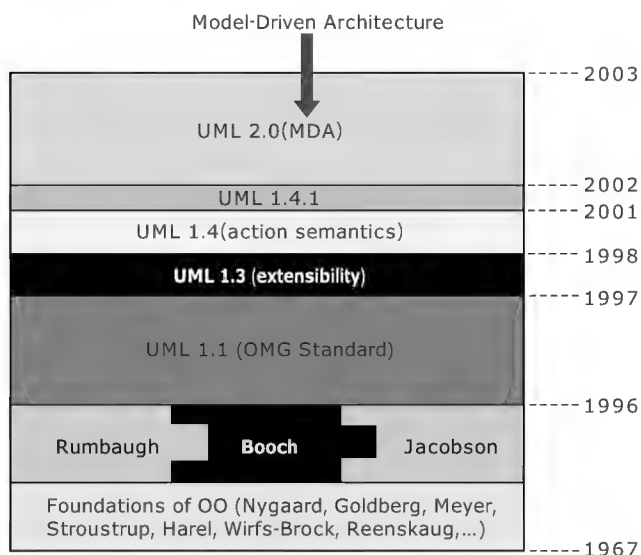
MDAでは，いったん取り去った詳細を取捨選択し，再びモデルに付加してモデルを進化させている．具体的には，UMLの各種ダイヤグラムがプログラム化している(**図3**，**図4**)．その結果，モデルがあればソースコードは不要になる．モデルを直接実行(executable UML ： exUML)すること，モデルから実装を自動生成(code generation)することが可能になる．MDAとexUML，コード生成は厳密には別の概念であるが非常に近く，一体化しやすい．`http://www.omg.org/`には，これらの概念に関連するドキュメントがいくつかあるが，簡潔で明確でわかりやすいものはないようである．

実際にこれらのこと(MDA，exUML，コード生成)を実現するためには，それぞれのアプリケーション分野独自の要素を柔軟に取り入れていくしくみが必要になる．このしくみは，ステレオタイプやプロファイルとして，以前からUML自身のアーキテクチャの中に組み込まれていた．

● リアルタイムプロファイル

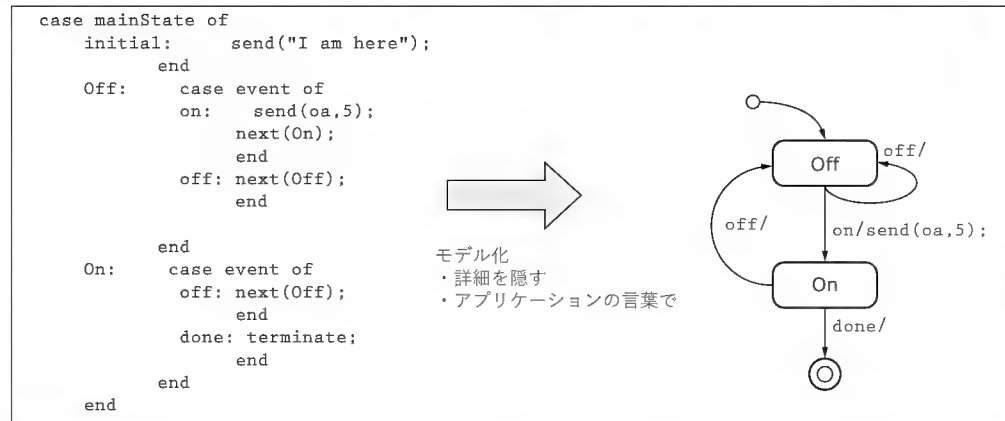
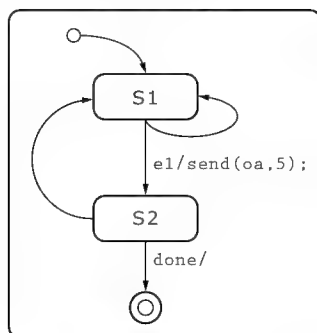
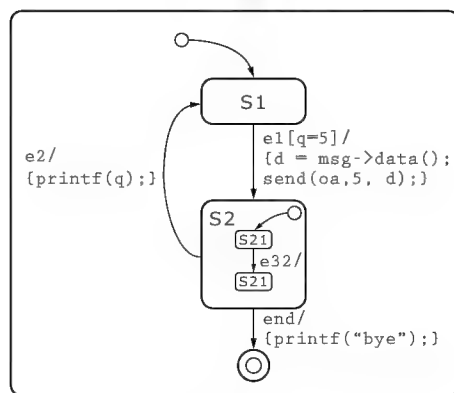
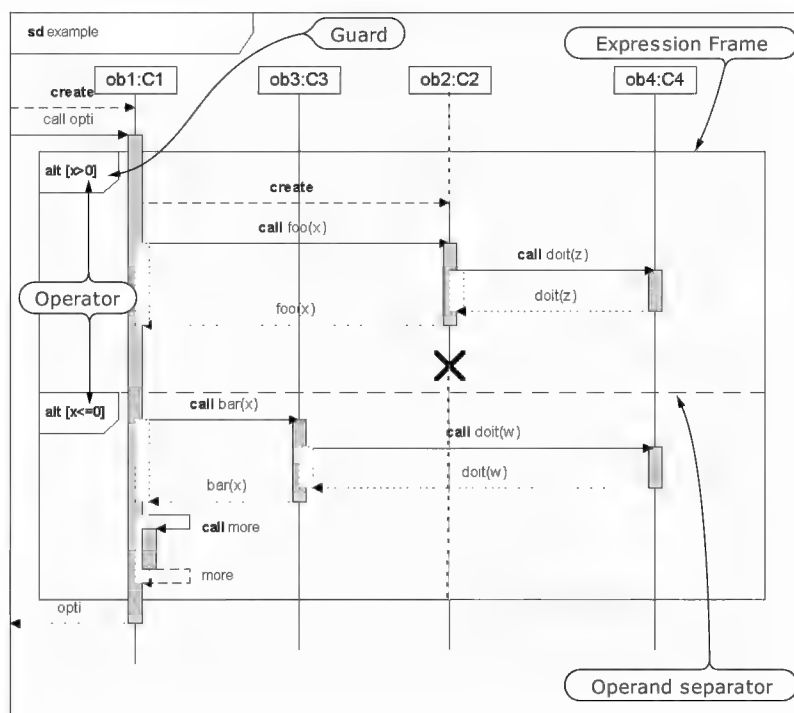
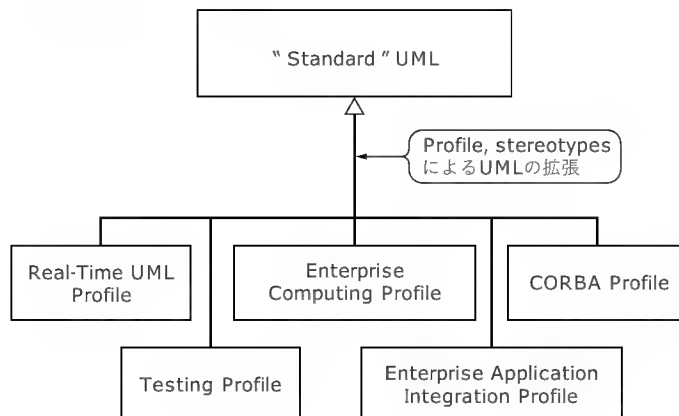
UML2.0では，組み込みリアルタイムシステム用にROOMのすべてとSDLの一部などがUMLに取り込まれた．ROOMという名前は，日本では組み込み用オブジェクト指向方法論としてよく使われるが，UML-RTが本当の名前である．UML-RT[注2]は，Rose RealTimeでexUMLを実現するために作成されたリアルタイムプロファイルである．このUML-RTで定義されていたモデル要素がUML本体(standard UML)に取り込まれることになる．

普通，**図5**を見ながらリアルタイムプロファイルというと，「UML Profile for Schedulability, Performance and Time」[注3]のことを指す．しかし，UML-RTのようなツールベンダーが

〔図1〕UMLの進化

注1：Selic, B., Gullekson, G., Ward, P., *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, NY, 1994.

注2：`http://www.rational.com/products/whitepapers/442.jsp`よりダウンロード可能．

注3：`http://www.omg.org/cgi-bin/doc?ptc/2002-03-02`よりダウンロード可能．

〔図2〕ソフトウェアのモデリング

〔図3〕モデルのプログラム化(1)

〔図4〕モデルのプログラム化(2)

自社製品をexUMLに対応させるために作成したものも含めてリアルタイムプロファイルと呼んでいる場合もある(**図6**).つまり,この意味ではTau(Telelogic),Rhapsody(i-Logix),Bridgepoint(Project Technology)など,それぞれ独自のリアルタイムプロファイルを,モデルを実行可能とするためにもっているといえる.**図7**のSDLプロファイルはこちらの意味である.

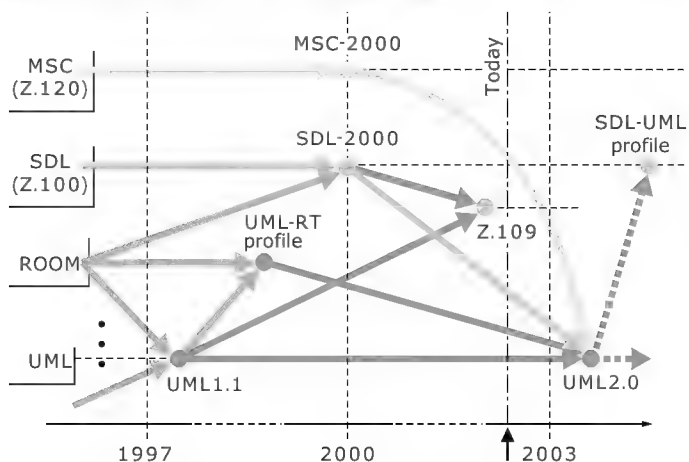
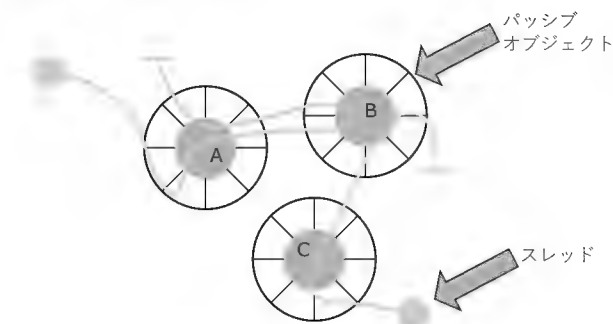
SDLの場合,UMLに取り込まれるのはその部分であり,残念ながらUMLをフロントエンドの分析言語として使用し,実装はSDLで行う形になる.この結果,オブジェクト指向では必須な繰り返し型の開発プロセスを取りにくいのではないかと思われる.SDLからUMLへリバースできない部分が存在するのがそ

〔図5〕UMLプロファイル

〔図6〕リアルタイムプロファイルの主要な経緯

〔図7〕SDLとUMLの関係

〔図8〕アクティブオブジェクトの表現

(UML 1.4 with Action Semanticsより)

〔図9〕オブジェクトとスレッド

の理由である．

今回は，UML2.0のアクティブオブジェクトモデリングに注目する．アクティブオブジェクトを使用すると，マルチスレッドを使用する組み込みリアルタイムシステムが作りやすくなる．

アクティブオブジェクトとは

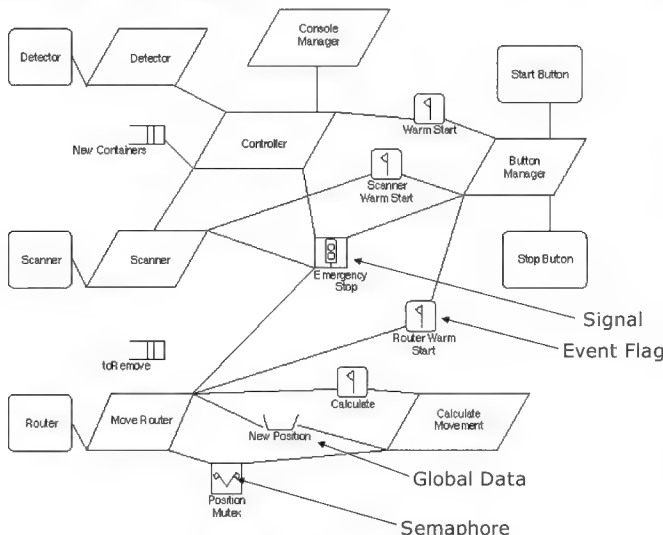
太線で描いたオブジェクトがアクティブオブジェクトである，といわれて納得していては話が始まらない．これは単なる表記法である．セマンティクスも規定するのがUMLである．アクティブオブジェクトのセマンティクス定義は，UML1.4以前のUMLではプロセスまたはスレッドのことである．たとえば，UML1.3の22章には，アクティブオブジェクトについて制御アクティビティを開始できるプロセスまたはスレッドと明記してある．スレッドをオブジェクトととらえるなら，それはRTOSの内部データのTCB(タスクコントロールブロック)あたりではないだろうか．TCBに`create()`とか`sleep()`などのシステムコールをオペレーションとしてもたせれば，オブジェクトになる．しかし，そういうRTOS内部の話ではなく，UML1.3がいっているのは，アプリケーションレベルのタスクを太線の箱で表すということである．

図8は*UML 1.4 with Action Semantics*[注4]からの引用である．この中のFactoryManager，Robot，Ovenのインスタンスがアクティブオブジェクトであり，スレッドに対応するものとして解釈されている．アプリケーションレベルのタスクあるいはスレッドは，プログラム内を走っているいくつもの線(スレッド)ととらえるのが妥当で，オブジェクトのように表現するのはMDAの観点からは無理があるように思う．

オブジェクトとスレッドについて**図9**に示す．

スレッドを箱として表現するUML1.3のアイデアは，コンカレント図(あるいはタスク図)というダイヤグラムで，Real-time Studio(ARTiSAN)などで利用されている(**図10**)．コンカレント図は，UMLのコラボレーション図を拡張したものに対応する．スレッドを，太線オブジェクトではなくひし形の箱で表現する．Gomaa氏のDARTS[注5]で見慣れたタスク間通信路記号のほかに，グローバル変数やセマフォなどもアイコンで表現する．複数のタスク間のある瞬間の関係を表現することができるので，タスク分割を検討する際の手助けになる．しかし，スレッドとオブジェクトの関係を表現できないのでこのまま動かすことはで

注4：`http://www.omg.org/cgi-bin/doc?ptc/2002-01-09`よりダウンロード可能．**図8**と同様の図は『UMLリファレンスマニュアル』(翻訳あり：ピアソン・エデュケーション，2002)にもある．
注5：H. Gomaa, *Software Design Methods for Concurrent and Real-time Systems*, Addison-Wesley, 1992, ISBN 0-201-52577-1.

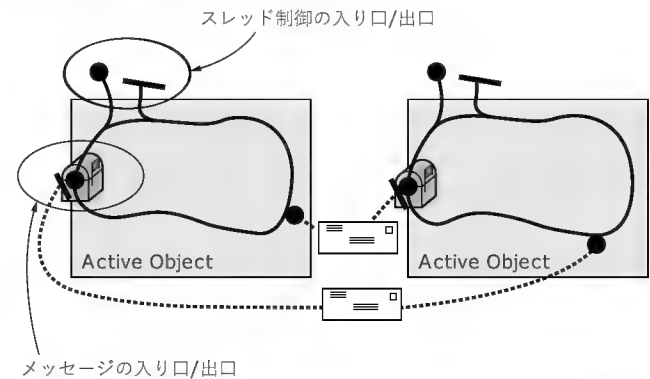
〔図10〕コンカレント図の例

(http://www.ddj.com/documents/s=913/ddj9812g/9812g.htmより)

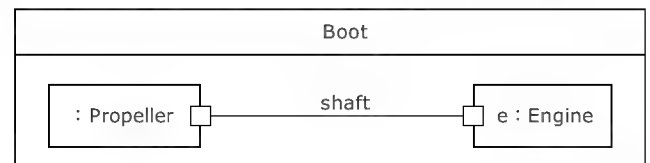
きない．スレッドとオブジェクトの関係とは，**図8**でFactory Managerアクティブオブジェクトの中にcurrentJobオブジェクトが入っているというような情報である．したがって，MDAを実現することは難しい．

UML1.4から，アクティブオブジェクトは，それ自身のスレッド制御をもっているオブジェクトという表現になった．しかし，これでも意味不明なので，UML2.0ではもう少し説明してある．生成すると仕様にしたがって動き出し，仕様を完了するか外部から止められるまで止まらない．これがそれ自身のスレッド制御をもつということである．タスクのメイン関数と同じである．**図11**に，UMLの抜粋とRose RealTimeにおけるアクティブオブジェクトの概念を示す．

Rose RealTimeのアクティブオブジェクトは内部にスレッド制御の閉じたループをもっている(スレッドではなくスレッドの制御である)．そのスレッド制御はランタイムフレームワークからやってきてまたランタイムフレームワークに帰るだけで，**図9**のように(アクティブ)オブジェクト間を渡り歩くことはしない．このことで，それ自身のスレッド制御をもっているというセマンティクスの実装を実現している．RTOS環境下で，タスクの制御がRTOSからやってきてRTOSに帰るのと同一である．タスクのメイン関数にあたる部分はランタイムフレームワークの中にある．つまり，アプリケーション層とフレームワーク層に明確に分かれている．

このような実装であれば，アプリケーション層とは独立に一つのメイン関数に対して複数のアクティブオブジェクトを対応させるメカニズムを実現することができる．このようにしないと，アクティブオブジェクトモデリングではタスク数が増えすぎてしまう．並列動作が前提のアクティブオブジェクト間の通信は，関数呼び出しでは実現できないのでメッセージパッシング

〔図11〕UML-RTアクティブオブジェクト

(UML 1.4より)
An Object has its own thread of control and runs concurrently with other active Objects. Such a class is informally called an active class.
(UML 2.0(draft)より)
An active object is an object that, as a direct consequence of its creation, ommences to execute its behavior specification, and does not cease until either the complete specification is executed or the object is terminated by some external agent. (This is sometimes referred to as" the object having its own thread of control ")

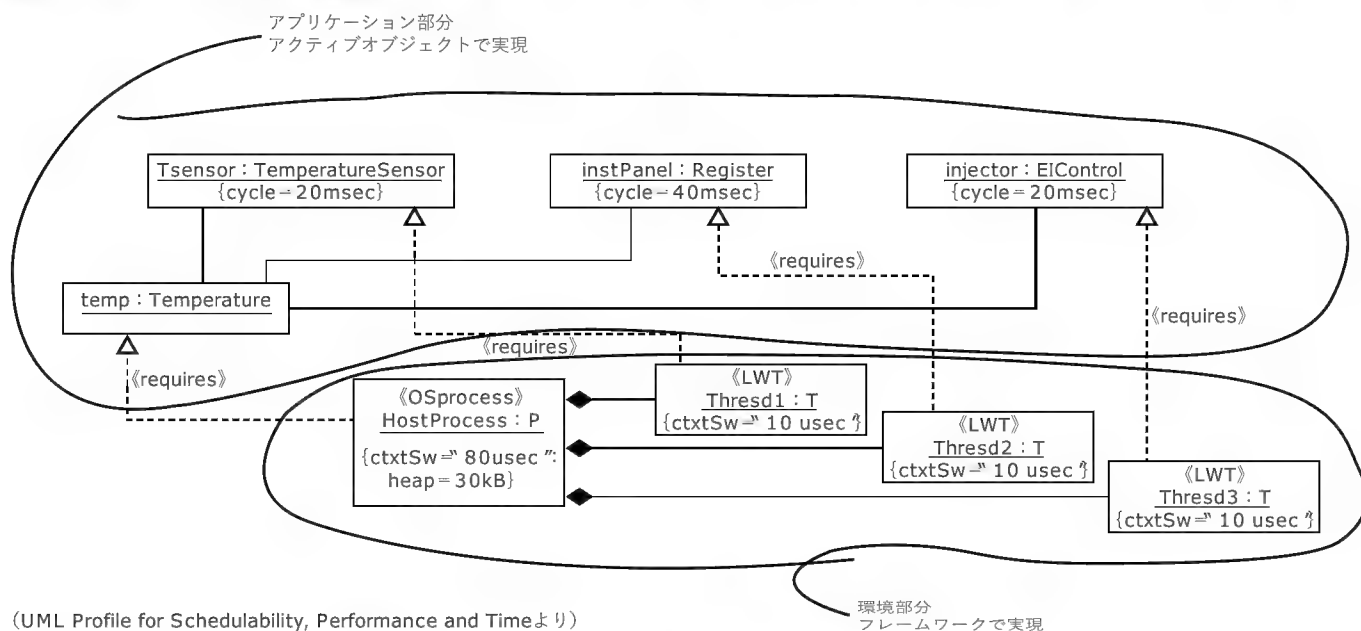
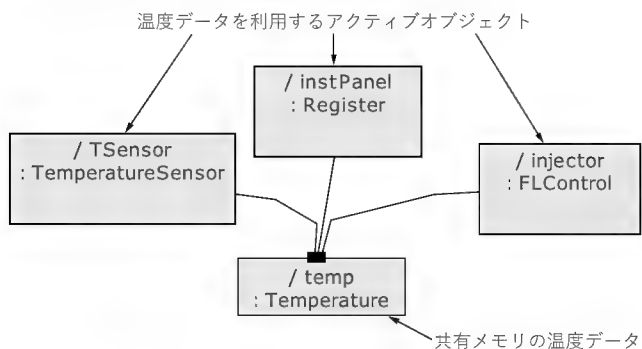
〔図12〕ポートによるオブジェクト間結合

(UML2.0(draft)より)

になる．**図11**の破線のスレッド制御は，ランタイムフレームワークによりメッセージが配信されるようすを表している．このような通信機構をUML2.0では，ポートとコネクタによって表現するようになる(**図12**)．これは，Rose RealTimeではカプセル構造図に対応する．

ランタイムフレームワークという言葉が出てきたが，マルチタスク環境でRTOSが必要なようにアクティブオブジェクトでモデリングを行い，そしてMDAを実現するためにはランタイムフレームワークが必要になる．

実際に動くものを開発するためには，アプリケーション部分を実現するだけでは不十分で，環境部分を作成しなければならない．シングルタスクのバッチ的に動くアプリケーションであれば，UNIXやWindowsといった共通のプラットホームを利用するのでほとんど意識しないが，独自ハードウェアで動作する組み込みシステムでは環境部分をどうするかが重要になる．**図13**の例では，一つのプロセスが三つのスレッドを所有する動作環境が表現されている．実際に動くモデルとするためには，このようなアプリケーション部分と動作環境部分の表現が必要である．組み込みでオブジェクト指向を導入すると，アプリケーション部分のモデリングに集中して環境側がおろそかになりが

〔図13〕アプリケーション部分と環境部分

(UML Profile for Schedulability, Performance and Timeより)

〔図14〕アプリケーション部分のみのモデル

ちである．そして，最後に統合化する段階で行き詰まることが多い．

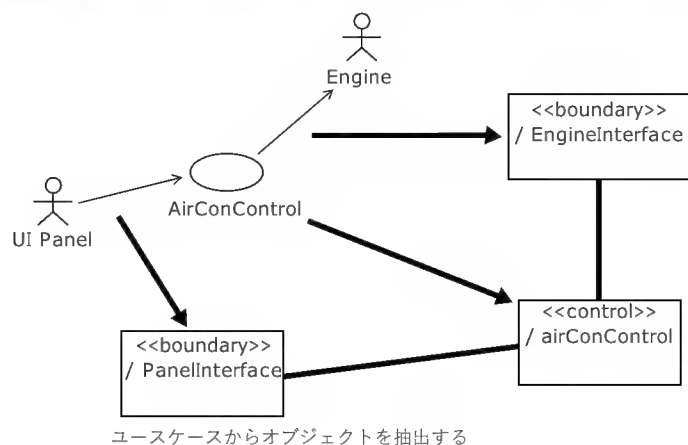
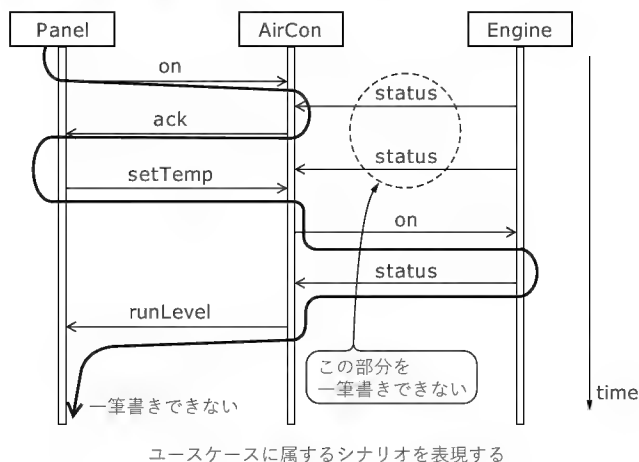
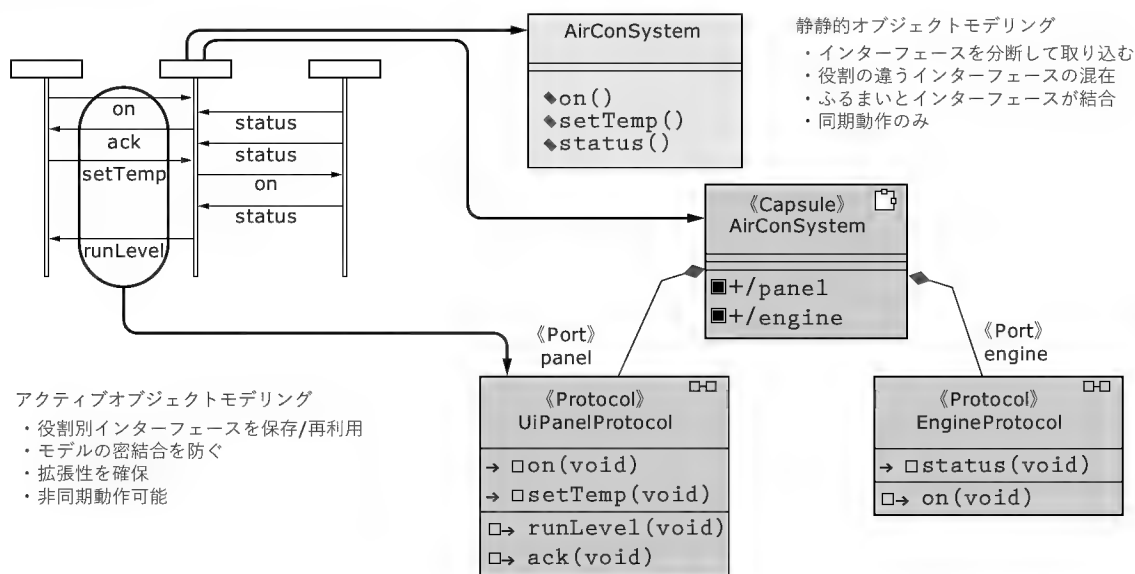
アプリケーション部分をアクティブオブジェクトでモデリングするのであれば，動作環境部分はフレームワーク化してプロセスやスレッドなどのRTOSの概念を隠蔽して汎用性をもたせることが可能になる．逆にそのようなフレームワークがあってはじめてアクティブオブジェクトモデリングが可能になる．**図10**のようなコンカレント図は，環境部分とアプリケーション部分それぞれを中途半端に表現するだけになってしまう．また，アクティブオブジェクトモデリングをうたっている場合でも，UML2.0以前の実態は単なるRTOSのラッパ程度のフレームワークしか提供していないものなど，さまざまであった．UML2.0では**図13**のアプリケーション部のみの表現は**図14**のようになる．アプリケーション部としてはここまでしか描かないので，環境部との分離は明確になる．その結果，環境部をフレームワーク化しやすくなる．

2 アクティブオブジェクトモデリング

すでにフレームワークが存在する場合には，アクティブオブジェクトモデリングは利用するフレームワークの影響を大きく受ける．ここでは，例としてRose RealTimeの場合を簡単に紹介する．別の環境ではまったく違う方法になる．

一般論として，オブジェクト指向ではユースケースによるシステム分析からモデリングを始める．この部分は，アクティブオブジェクトもパッシブオブジェクトもかわりない．次に，分析クラスを使ってクラス抽出を行う．ここも，基本的にかわりない．使用する方法論によってクラス抽出の仕方がかわるかもしれないが，影響は受けない．

次に，ユースケースのシナリオ一つ一つを抽出したクラスを使ってシーケンス図で表現する．この作業の中で，分析クラスを実際の設計クラスに置き換えつつクラス抽出の見直しを行いながら，クラスを分割したり，統合する(**図15**)．このあたりからモデリングの違いが出てくる．パッシブクラスの場合，クラス間インターフェースは関数呼び出しなので，**図16**のようなマルチスレッドシーケンス図の実装可能性確認はできず，シングルスレッド動作まで分割しなければならない場合が多い．

シングルスレッドのシーケンス図とは，一筆書きできるアミダくじのようなシーケンス図である．このときの一筆書きの線がスレッド制御に対応する．**図16**では，Engineから周期的にstatusを送信してくることを表現しているので一筆書きができない．つまりこのままでは，パッシブオブジェクトモデリングはできない．しかし，アクティブオブジェクトモデリングでは，オブジェクトはそれぞれ独自のスレッド制御をもっているので，メッセージ線を伝わってスレッド制御が移動することはない．そ

れぞれのアクティブオブジェクトは並列に動作することが前提なので，シングルスレッドまで分解する必要はない．パッシブオブジェクトモデリングでは，同期動作するところまで分割しなければならない．同期動作まで分解するためにはタスク分割が必要になる．しかし，タスク分割するにはこの段階では情報が不足していることが多い．したがって，パッシブオブジェクトモデリングでは前にもどる覚悟で先に進めることになる．

アクティブオブジェクトモデリングではタスク分割はフレームワーク側に分離されているので，前にもどる心配はない．シュレイアー・メラー法の言葉でいえば，問題領域(ドメイン)が分割されている．

ユースケースのシナリオすべてのシーケンス図が描けたら，それぞれのクラスのインターフェースを決めることになる．インターフェースの扱いは決定的に違ってくる．Rose RealTimeのアクティブオブジェクトモデリングでは，アクティブオブジェクトのインターフェースは，アクティブオブジェクトとは別のクラスで実現する(**図17**)．パッシブクラスのAirConSystemは，自分への入力メッセージに対応したオペレーション，on()，setTemp()とstatus()をもつことになる．この中で，onとsetTempはユーザーインターフェースのコントロールパネルからの入力であり，エアコンのインバータ側からのstatus入力と一緒にされる必然的な理由はない．たまたま行ったクラス抽出の結果，本来一緒にされるべきonとsetTemp，ack，runLevelと切り離され，本来なら別々であるstatusと一緒になっているだけである．しかも，拠り所のクラス抽出は，将来のタスク分割の結果によって見直される可能性が高いのである．

一方，インターフェースを別クラスで実装するRose RealTime型のモデリングでは，役割別のインターフェースは役割別にそのまままとめておくことができる．そして，インターフェースとふるまいを明確に分割して実装できる．このことは，モデルの

理解しやすさや再利用性を向上させる．しかも，タスク分割の影響を受けないので非常に安定したモデルを初期の段階から構築できる．マルチスレッド環境でのアクティブオブジェクトモデリングの優位性は，疑う余地がない．

Rose RealTime のランタイムフレームワークの原型である ROOM については難しいという話をよく聞くが，先に説明したように実際は非常に簡単である．むしろ，パッシブクラスで実装するほうが困難である．とくに，大きなシステムになるほどクラス分割とタスク分割を同時に考えながら設計するのは至難の業である．ROOM が難しいというのはほとんど誤解である．

誤解の原因は，ROOM を解説した分厚い本にあるのではないかと思う(**図 18**)．この本のほとんどの部分はランタイムフレームワークの話である．アプリケーションを作るために必要な部分は，最後の三つの章だけである．第 12 章が設計モデリングに関すること，第 13 章がアーキテクチャモデリング，第 14 章が開発プロセスに関することである．この本を予備知識なしに最初から読むと挫折する．パッシブクラスを使ってフレームワーク自身を作りたい人(すなわち**図 13** のレベル)は最初から読む必要があるが，Rose RealTime を使用してアプリケーションのみを作る人(すなわち**図 14** のレベル)は，第 12 章から読めば十分である．逆に他の部分を読むと混乱すると思う．RTOS の作り方を学ぶのと RTOS の使い方を学ぶことの違いに相当する．

● タスク分割の自動化

タスク分割とアクティブオブジェクトモデリングの関係は，どうなるのであろうか．アクティブオブジェクトを使用した場合，一つのタスクに複数のアクティブオブジェクトを載せるようになる．タスク生成などの具体的な実現の仕方はフレームワークの中で解決済みなので，設計者の仕事はアクティブオブジェクトとタスクとのマッピングを指定するだけになる．そして，マッピングの取り方はレートモノトニック分析により，いまでは自動化されている．数年前までは考えられないことであるが，スキルの要求されていたタスク分割は，いまでは自動化される時代になったのである．UML を使用するメリットは，実行可能モデルを作るだけではなく，モデル解析ツールと連携することによりモデルの検証ができることである(**図 19**)．将来は，タスク分割だけでなく前回扱った時間オートマタによる仕様の検証なども可能になるだろう．

自動タスク分割の手順は，まずデッドライン付きの外部イベントを洗い出す．そして，その外部イベントそれぞれについて処理手順をシーケンス図で表現し，各処理の最悪実行時間を指定する．そして，ツールを起動する．

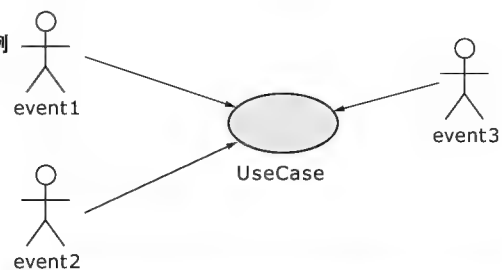
たとえば簡単な例として，デッドラインつきの三つのイベント(event1，event2，event3)があったとする(**図 20**)．このときに，イベントの到着パターンも指定する．例では周期イベントにしてあるが，分布関数による指定も可能である．

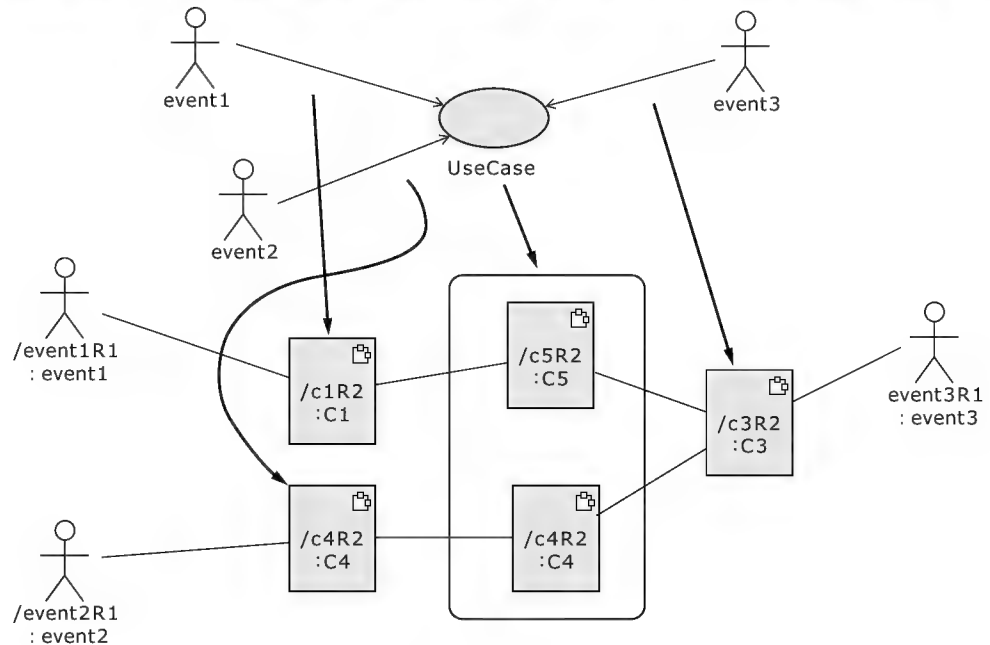
次に，オブジェクト抽出を行う．ここでは，ユースケースから二つのオブジェクトが抽出されて全体で五つのアクティブオブジェクトが抽出されたとする(**図 21**)．

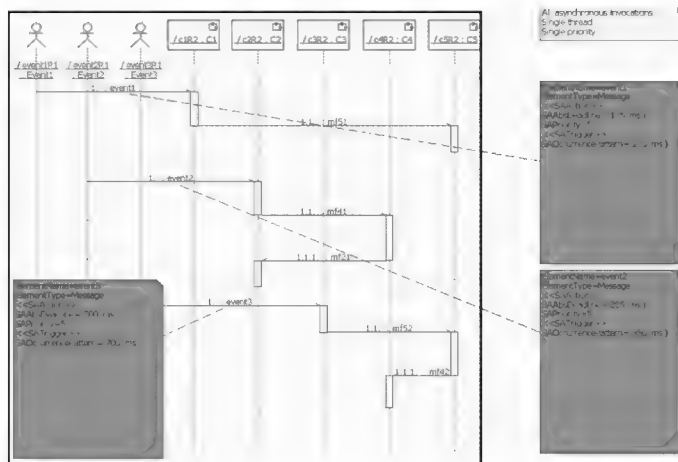
次に，抽出したオブジェクト間のメッセージパッシングにより，外部オブジェクトの処理手順をシーケンス図で記述する．**図 22** では，簡単のために一つのシーケンス図にまとめてしまったが，本当は別々に描く．また，メッセージの種類も非同期メッセージにしてあるが，目的に応じて同期メッセージ，関数呼び出しを指定することができる．メッセージの種類もタスク分割に反映される．つまり，同期メッセージか関数呼び出しであれば同一スレッドにマップされる．非同期メッセージの場合は，時間制約とリソース制約によって別スレッドになるか同一スレッドになるかをツールが決定する．また例では，各イベント処理が一筆書きまで落とされているが，非同期メッセージを使う場

〔**図 18**〕**ROOM 本**

〔**図 19**〕**モデル解析ツールとの連携**

〔**図 20**〕
外部イベント例

Input Event	デッドライン	起動周期
Event1	135	270
Event2	285	350
Event3	700	700

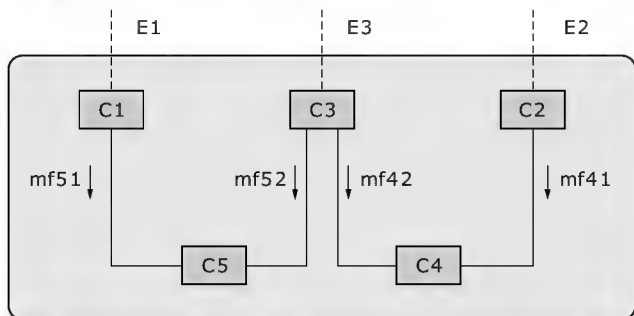
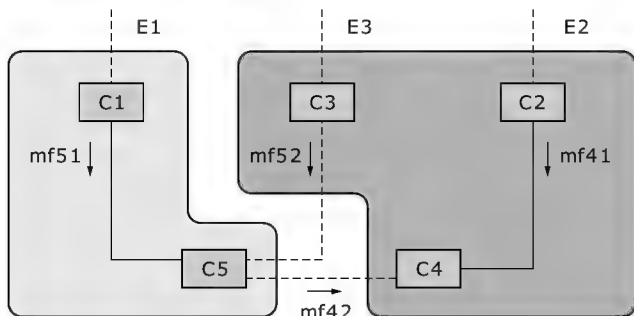
〔図21〕オブジェクト抽出

〔図22〕シーケンス図による表現と時間制約の指定

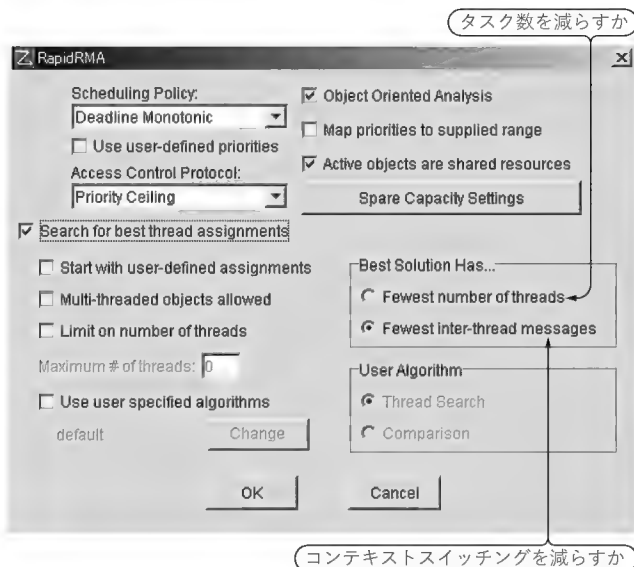
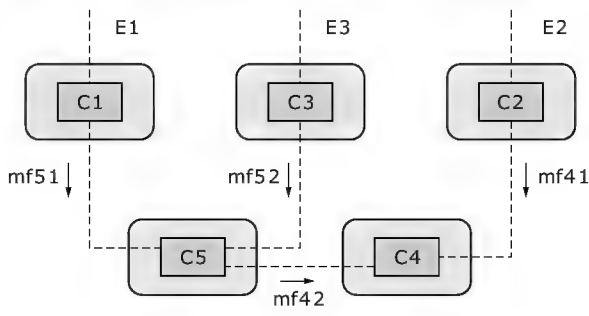
〔図23〕イベントごとにスレッドを割り当てた場合

合，途中でスレッド制御が分岐するような処理ももちろん，実現可能かつ解析可能である．

非同期メッセージ送信を直接RTOSのメールボックスなどのシステムコールによる実装に落とすと，そこはタスク境界になってしまい，タスク分割を行っていることになる．それではコンカレント図を使用する分析設計やOCTOPUS法と同一になってしまう．RTOSが提供するのはタスク間メッセージ通信のみであり，タスク内で使用することは想定されていないので，同一タスク内で使用するとデッドロックになったりする．したがって，たとえば後から，リソース不足などでタスクを結合させる際に非同期から同期へのメカニズムの変更が必要になってしまう．スレッドマッピングの自由度を確保するためには，タスク間でもタスク内でも同様に使用できる非同期通信メカニズムをフレームワークで提供する必要がある．

一般に人間がタスク分割を行うと，**図23**のようなイベントごとのタスク分割になりがちである．そして，開発の後半で実機によって試験した際に時間制約を守れないと，C_4やC_5の処理時間を短縮することで何とか乗り切ろうとする．しかし，それだけでは選択肢が少なすぎて対応できないこともある．また，C_4やC_5の中にハードウェアに依存しすぎる痕跡を残すことになり，部品性を損なうことになる．また，最悪の場合，モデル構造全体が崩壊することもある．

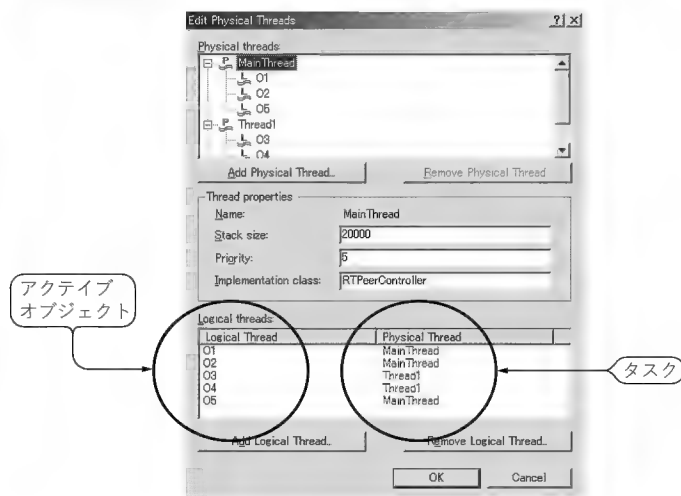
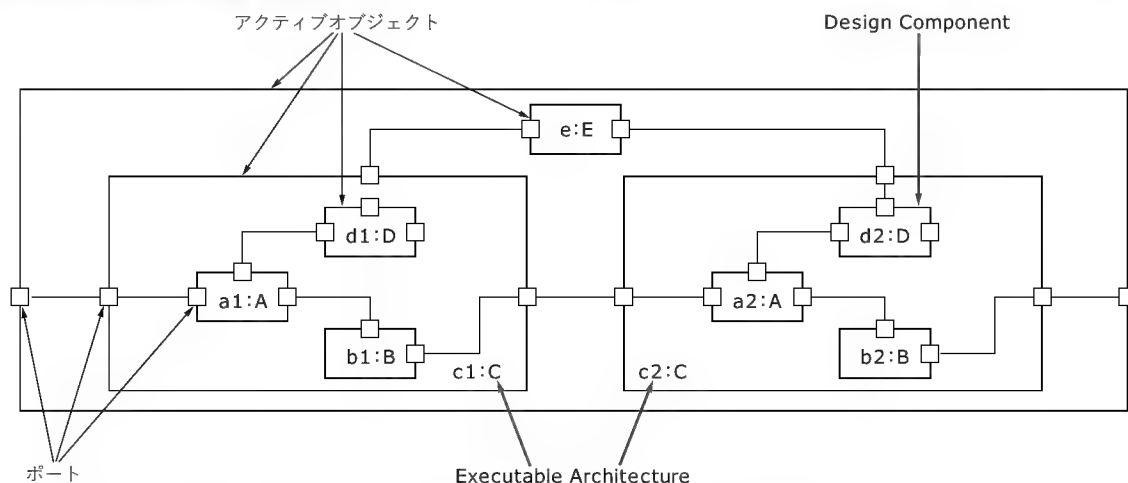
実際は，モデルを崩壊させる前に**図24**～**図26**に示すような各種の構成を検討するべきである．このような組み合わせを自由に実施できるところが，アクティブオブジェクトモデリングの大きな利点である．また，作成した組み合わせをツールにより検証できることと，組み合わせ自身をツールが生成して自動化できることも重要である．ただ，このスレッドマッピング問題はいわゆるNPハード問題であり，アクティブオブジェクト数が多くなると組み合わせ数が指数関数的に増えるので最適解を検索できない．しかし，実行可能解を見つけることはできる．タイミング解析ツールが提供する解は，実行可能解である．

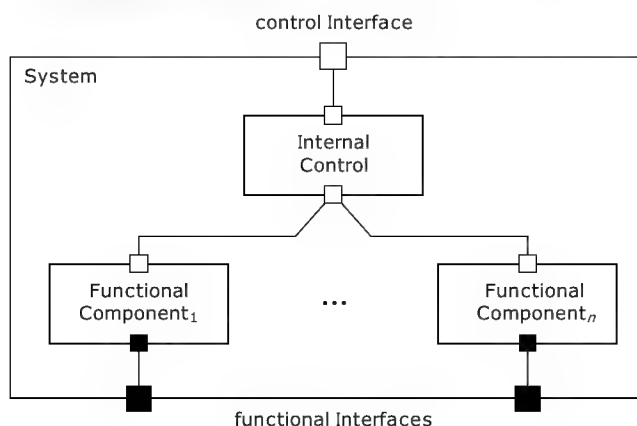
〔図24〕シングルスレッドの場合

〔図26〕二つのスレッドで実現した場合

RapidRMAによるタイミング解析パラメータの設定画面を**図27**に示す．スケージューリング方式やリソース同期方式ばかりでなく，最適化オプションとして，タスク数を減らすか，タスク間通信を減らすか，どちらかを指定できる．タスク数を減らせばスタックなどのRAMを節約できる．一方，タスク間通信を減らせばコンテキストスイッチングが減るので，実行時間を短縮できる．必要なパラメータを設定してOKをクリックすれば，タスク分割が行われる．そしてタスク分割の結果は，自動的にRose RealTimeのタスクマッピング仕様として**図28**のように取り込まれる．

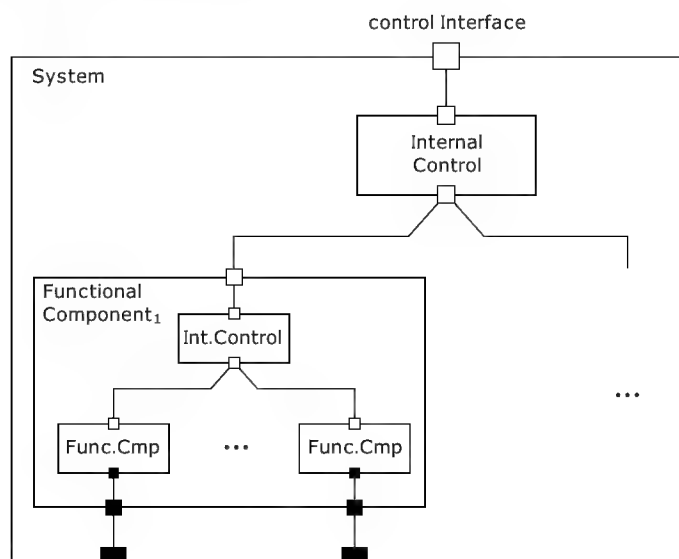
〔図27〕RapidRMAによるタイミング解析

〔図25〕オブジェクトごとにスレッドを割り当てた場合

3 構造化クラス

アクティブオブジェクトの中に，アクティブオブジェクトを入れることができる．これは，UML1.3レベルのアクティブオブジェクトではできないことである．オブジェクト間の全体部分の関係はクラス図で表現される．UML2.0ではこれとは別に，構造化クラスという概念を導入している．構造化クラスは，内部にコラボレーション図をもつ．クラス図の全体部分の関係だけでは，ポートとコネクタによって表現される通信関係を表現することができないので，構造化クラスはコラボレーション図で内部の接続構造を表現する．

アクティブオブジェクトの中にアクティブオブジェクトを入れるというのは，アクティブオブジェクトが構造化クラスであり内部のコラボレーション図で全体部分の関係にある別のアクティブオブジェクトとの接続関係を表現できるということである(**図29**)．前節までの話では，一つのアクティブオブジェクトを設計要素として見てきた．しかしアクティブオブジェクトは構造化することで，アーキテクチャを表現することも可能である．このことは，アクティブオブジェクトモデリングのもう一つの重要なポイントである．

〔図28〕RoseRTによるアクティブオブジェクトとタスクとのマッピング

〔図29〕アクティブオブジェクトの組み合わせ

〔図30〕機能部分と制御部分を別のオブジェクトとして実装

〔図31〕さらに入れ子にする

この連載の第6回(本誌2002年11月号)の最後に，状態マシン設計に関連して，主要機能レベルと補助機能レベル，制御レベルで状態マシンを階層化することについて述べた．しかし，主要機能レベルの要求はユースケースからもたらされ，補助機能レベルはユースケースのような機能ベースの分析からではなく通信路の信頼性確保などの非機能的要求からもたらされる．また，制御レベルはハードウェアの不具合回避などの要求にもよる．このように，それぞれ役割の異なるものは，階層化だけでは分離が不十分で，別のオブジェクトにすべきであると述べた．ふるまいとインターフェースを分離した構造化可能なアクティブオブジェクトを利用することで，このことが可能になる．ふるまいとインターフェースが結合し，クラス図のみで構造化されていないパッシブクラスでは，対応は難しい．

たとえば，**図30**のようなアーキテクチャが可能になる．組み込み系の場合，制御部分の仕様追加・変更が頻発する傾向があるが，そのような場合でも，**図31**に示したようにアーキテクチャの入れ子構造で対応可能になる．

おわりに

UML2.0の話題から，Rose RealTimeのアクティブオブジェクトモデリングを紹介した．アクティブオブジェクトモデリングでは，フレームワークが目的のアプリケーションに適合するかどうかが重要になる．ツールを選定する際に，ツール機能の比較表を作成してあれができる，これができないと詳細に検討することがある．このとき重要なのは，一つ一つの機能の有無ではなく，どのようなフレームワークが利用できるかである．

ふじくら・としゆき　日本ラショナルソフトウェア(株)

注6：緊急報告：本連載は継続中ですが，藤倉氏に寄稿いただいたこれまでの本誌掲載記事を中心に加筆・再編集し，本誌増刊TECH I vol.15として2002年12月14日に発売予定です．ご期待ください！　(編集部)

x86CPUだけでもマスタしたい

開発技術者のためのアセンブラ入門

第⑭回 CPUのデータ転送(その2)

大貫広幸

今回は，前回説明できなかったi486およびPentium以降で使用できるデータ転送命令について解説します．

アセンブラでのデータ転送命令の記述2

ここでは汎用命令のうち，i486以降，Pentium以降といった使用可能なCPUを選ぶデータの転送命令と，分類上「その他の命令」にある転送命令について説明します．

表1は，今回説明する転送命令を示したものです．

● BSWAP命令

BSWAP命令は，i486以降で使用できる命令です．

BSWAP命令は，オペランドで指定された32ビットレジスタ内の各バイト値を交換するための命令です．

交換されるバイトは，**図1**のように，

(ビット31～24)⟷(ビット7～0)

(ビット23～16)⟷(ビット15～8)

です．

この命令は，ビックエンディアンで表される値をリトルエンディアンの値に変換する場合や，その逆にリトルエンディアンで表される値をビックエンディアンに変換する場合に使用します．

実際のMASMでの記述例を**リスト1**，gasでの記述例を**リスト2**に示します．

● XADD命令

XADD命令も，i486以降で使用できる命令です．

XADD命令は転送と加算を一つにした命令で，転送先をDEST，転送元をSOUで表した場合，

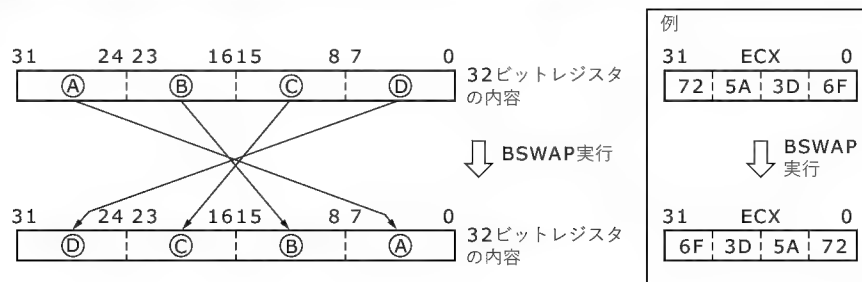
- まずDESTをSOUに転送
- 次にDESTとSOUの元の値を加算した値をDESTに転送

と動作します．この場合，加算を実行しているため，フラグは次回解説する予定のADD命令と同じ影響を受けます．

〔表1〕i486，Pentium以降の32ビットCPUで使用できる転送命令，および「その他の命令」に分類されている転送命令

分類	インストラクション名	動作	影響を受けるフラグ	使用可能CPU
データ転送命令	BSWAP	Byte Swap ● 32ビットレジスタ内のバイトを交換し，リトルエンディアン⟷ビックエンディアンの変換を行う	なし	486以降
	XADD	Exchange and Add ● DESTをSOUに転送し，元のSOUとDESTを加算しDESTに設定する	CF，PF，AF，ZF，SF，OFが加算結果にしたがい設定される	486以降
	CMPXCHG	Compare and Exchange ● アキュムレータ(AL，AX，EAX)とDESTを比較し，等しければ，ZF←1，DEST←SOUを行う 等しくなければ，ZF←0，アキュムレータ←DESTを行う	CF，PF，AF，ZF，SF，OFが比較結果にしたがい設定される	486以降
	CMPXCHG8B	Compare and Exchange 8 Bytes ● EDX：EAXとDESTを比較し，等しければ，ZF←1，DEST←ECX：EBXを行う 等しくなければ，ZF←0，EDX：EAX←DESTを行う	CF，PF，AF，ZF，SF，OFが比較結果にしたがい設定される	Pentium以降 CPUID命令で使用可能か調べる
	CMOVcc	Conditional Move ● ccで示された条件が成立する場合に，DEST←SOUの転送を行う	なし	Pentium Pro以降 使用可能かCPUID命令で調べる
その他の命令	LEA	Load Effective Address ● SOUの実効アドレス(オフセット)をDESTのレジスタに設定	なし	すべてのCPU
	XLAT	Table Look-up Translation ● レジスタ(E)BXが示すメモリ上のバイトテーブルからレジスタALをインデックスとして値をレジスタALにロードする AL←〔DS：((E)EBX+符号なし整数としてのAL)〕	なし	すべてのCPU

● 表中のDESTはdestination(先)，SOUはsource(元)を表す

〔図1〕BSWAP命令の動作

このXADD命令の動作を図で表すと，**図2**のようになります．実際のMASMでの記述例を**リスト1**，gasでの記述例を**リスト2**に示します．

● CMPXCHG命令

CMPXCHG命令も，i486以降で使用できる命令です．

CMPXCHG命令は比較と転送を一つにした命令で，転送先をDEST，転送元をSOUで表した場合，

● まずアキュムレータ(AL，AX，EAX)とDESTを比較
● 比較の結果，アキュムレータ＝DESTなら，
　ZFは1となり，SOUをDESTに転送
● 比較の結果，アキュムレータ≠DESTなら，
　ZFは0となり，DESTをアキュムレータに転送

と動作します．この場合，比較を実行しているため，フラグは次回述べるCMP命令と同じ影響を受けます．

アキュムレータとしてレジスタALが使われるのか，レジスタ

〔リスト1〕MASMのBSWAP，XADD，CMPXCHG，CMPXCHG8B，CPUID命令の記述例

```
                                        .586
                                        .model flat

 00000000                               .data

 00000000 01                            dtByte   db      1
 00000001 0002                          dtWord   dw      2
 00000003 00000004                      dtDWord  dd      4
 00000007                               dtQWord  dq      8  ← 8バイト(64ビット)長のデータはDQで定義する
          0000000000000008

 00000000                               .code
 00000000  0F CB                                 bswap   ebx

 00000002  0F C0 1D                              xadd    dtByte,bl
           00000000 R
 00000009  66| 0F C1 15                          xadd    dtWord,dx
           00000001 R
 00000011  0F C1 05                              xadd    dtDWord,eax
           00000003 R
 00000018  0F C0 DC                              xadd    ah,bl
 0000001B  66| 0F C1 D6                          xadd    si,dx
 0000001F  0F C1 C7                              xadd    edi,eax

 00000022  0F B0 1D                              cmpxchg   dtByte,bl
           00000000 R
 00000029  66| 0F B1 15                          cmpxchg   dtWord,dx
           00000001 R
 00000031  0F B1 05                              cmpxchg   dtDWord,eax
           00000003 R
 00000038  0F B0 DC                              cmpxchg   ah,bl
 0000003B  66| 0F B1 D6                          cmpxchg   si,dx
 0000003F  0F B1 C7                              cmpxchg   edi,eax

 00000042  0F C7 0D                              cmpxchg8b    dtQWord
           00000007 R

 00000049  0F A2                                 cpuid

                                        end
```

〔リスト2〕gasのBSWAP，XADD，CMPXCHG，CMPXCHG8B，CPUID命令の記述例

```
  1                        .data
  2
  3  0000 01               dtByte:    .byte  1   ここで使用している
  4  0001 0200             dtWord:    .word  2   gasは，AT＆T表記の他に
  5  0003 04000000         dtDWord:   .long  4   インテル表記のニモニック
  6  0007 08000000         dtQWord:   .quad  8   も使用できた
  6       00000000
  7
  8                        .text
  9  0000 0FCB                        bswap   %ebx
 10  0002 0FCB                        bswapl  %ebx
 11
 12  0004 0FC01D00                    xadd    %bl,dtByte
 12       000000
 13  000b 0FC01D00                    xaddb   %bl,dtByte
 13       000000
 14  0012 660FC115                    xadd    %dx,dtWord
 14       01000000
 15  001a 660FC115                    xaddw   %dx,dtWord
 15       01000000
 16  0022 0FC10503                    xadd    %eax,dtDWord
 16       000000
 17  0029 0FC10503                    xaddl   %eax,dtDWord
 17       000000
 18  0030 0FC0DC                      xadd    %bl,%ah
 19  0033 0FC0DC                      xaddb   %bl,%ah
 20  0036 660FC1D6                    xadd    %dx,%si
 21  003a 660FC1D6                    xaddw   %dx,%si
 22  003e 0FC1C7                      xadd    %eax,%edi
 23  0041 0FC1C7                      xaddl   %eax,%edi
 24
 25  0044 0FB01D00                    cmpxchg    %bl,dtByte
 25       000000
 26  004b 0FB01D00                    cmpxchgb   %bl,dtByte
 26       000000
 27  0052 660FB115                    cmpxchg    %dx,dtWord
 27       01000000
 28  005a 660FB115                    cmpxchgw   %dx,dtWord
 28       01000000
 29  0062 0FB10503                    cmpxchg    %eax,dtDWord
 29       000000
 30  0069 0FB10503                    cmpxchgl   %eax,dtDWord
 30       000000
 31  0070 0FB0DC                      cmpxchg    %bl,%ah
 32  0073 0FB0DC                      cmpxchgb   %bl,%ah
 33  0076 660FB1D6                    cmpxchg    %dx,%si
 34  007a 660FB1D6                    cmpxchgw   %dx,%si
 35  007e 0FB1C7                      cmpxchg    %eax,%edi
 36  0081 0FB1C7                      cmpxchgl   %eax,%edi
 37
 38  0084 0FC70D07                    cmpxchg8b     dtQWord
 38       000000
 39
 40  008b 0FA2                        cpuid
```

〔図2〕XADD命令の動作

〔図3〕CMPXCHG命令の動作

AXが使われるのか，それともレジスタEAXが使われるのかは，オペランドで指定されたデータのサイズにより決まります．オペランドのサイズがバイトならAL，ワードならAX，ダブルワードならEAXとなります．

このCMPXCHG命令の動作を図で表すと，**図3**のようになります．

実際のMASMでの記述例を**リスト1**，gasでの記述例を**リスト2**に示します．

● CMPXCHG8B命令

このCMPXCHG8B命令は，Pentium以降で使用できる命令です．このCMPXCHG8B命令が使用できるか否かは，CPUID命令により判断できます．

CMPXCHG8B命令というのは，今述べたCMPXCHG命令を8バイト(64ビット)にしたものです．

転送元(SOU)は，二つのレジスタECX：EBXで表される64ビット値に固定されています．そのため，オペランドは転送先(DEST)のみ指定することになります．この場合，DESTはメモリ上のクワッドワード(8バイト，64ビット)を指定します．

実際のMASMでの記述例を**リスト1**，gasでの記述例を**リスト2**に示します．

動作は，

- まずEDX：EAXの64ビット値とDESTを比較
- 比較の結果，EDX：EAX＝DESTなら，
  ZFは1となり，SOUのECX：EBXをDESTに転送
- 比較の結果，EDX：EAX≠DESTなら，
  ZFは0となり，DESTをEDX：EAXに転送

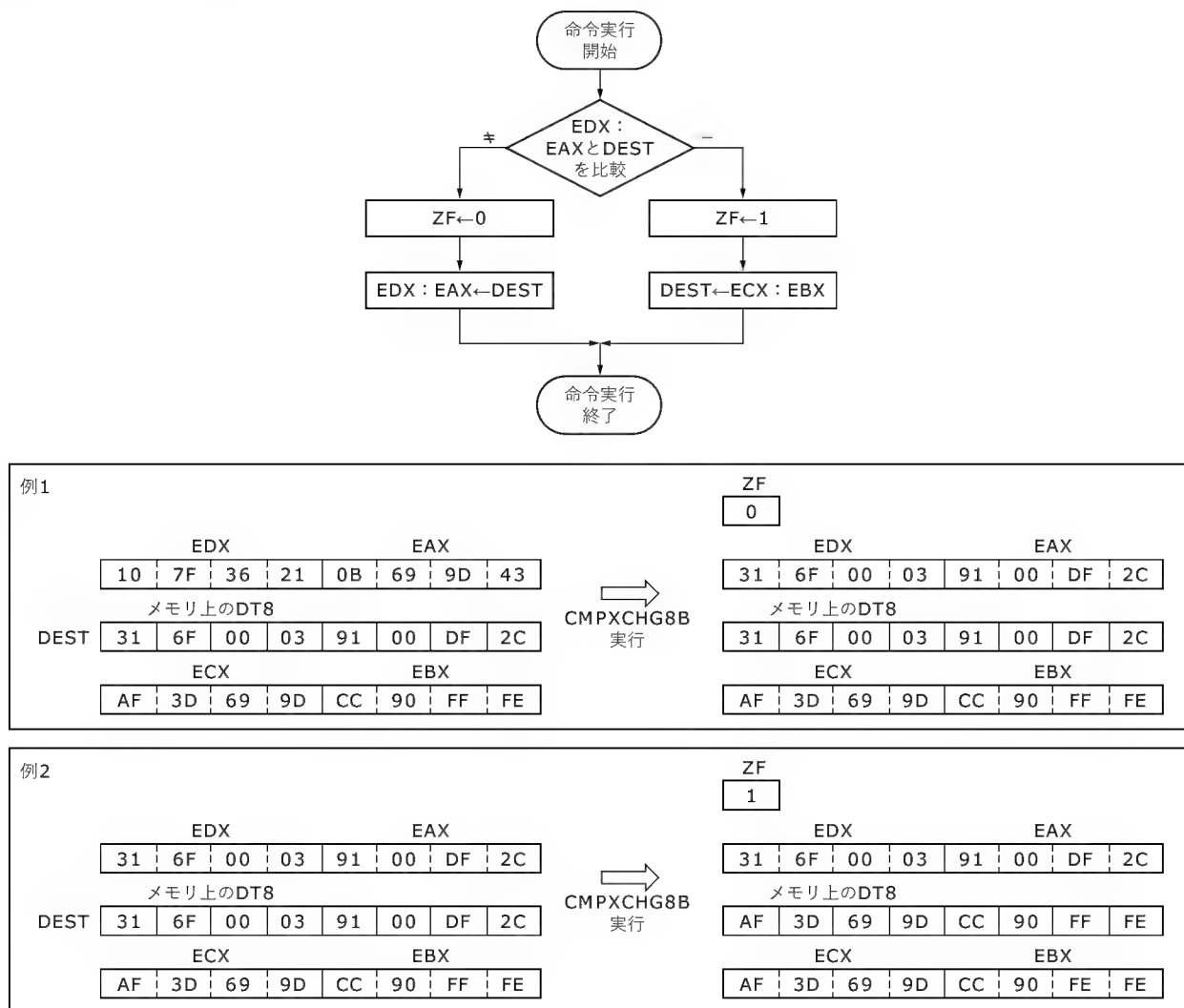
となります．

このCMPXCHG命令の動作を図で表すと，**図4**のようになります．

● CMOVcc命令

このCMOVcc命令はPentium Proから取り入れられた命令で，使用可能か否かはCMPXCHG8B命令と同じように，CPUID命令により判断します．

CMOVcc命令をMASM(Ver6.14)で使用する場合は，CPU指定を「.686」にします．gas(Ver2.10.90)はそのままでCMOVcc命令が使用できます．

〔図4〕**CMPXCHG8B命令の動作**

CMOVcc命令は条件付き転送の命令で，ccで示された条件が成立する場合にのみ転送が行われます(**図5**).

CMOVcc命令はインテル表記の場合,

```
CMOVcc   dest,sou
```

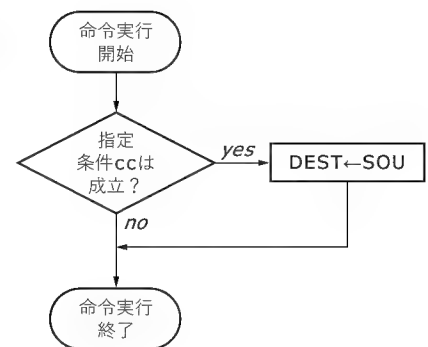
となります．ccの部分には**表2**の条件が入ります.

扱えるデータのサイズは，16ビット(2バイト)と32ビット(4バイト)の2種類です．destは16ビットあるいは32ビットの汎用レジスタ，souには16ビットあるいは32ビットの汎用レジスタか，メモリ上の16ビットあるいは32ビットの値を指定します.

● CPUID命令によるCMPXCHG8B命令，CMOVcc命令の使用判断

CPUID命令は，Pentium以降のプロセッサなら使用できる命令で，CPUに関する情報を取得することができます.

CPUID命令には，オペランドがありません(**リスト1**，**リスト2**)．その代わり，レジスタEAXに値を設定し，取得したい情報を指定します.

〔図5〕
CMOVcc命令の動作

CPUID命令の実行結果，つまりCPUの情報は，汎用レジスタのEAX，EBX，ECX，EDXの四つのレジスタに設定されてきます.

CMPXCHG8B命令やCMOVcc命令が使用できるか否かは，レジスタEAXに1を設定し，CPUID命令を実行することで，レジスタEDXに設定されてきます．レジスタEDXのビット8が1

〔表2〕ニモニックで使われる条件を表す文字

文字	内容	成立となる条件
E	Equal ...等しい	ZF=1
Z	Zero ...ゼロ	ZF=1
NE	Not Equal ...等しくない	ZF=0
NZ	Not Zero ...ゼロではない	ZF=0
A	Above ...より上	CF=0 and ZF=0
NBE	Not Below or Equal ...より下でなく等しくない	CF=0 and ZF=0
AE	Above or Equal ...より上か等しい	CF=0
NB	Not Below ...より下でない	CF=0
B	Below ...より下	CF=1
NAE	Not Above or Equal ...より上でなく等しくない	CF=1
BE	Below or Equal ...より下か等しい	CF=1 or ZF=1
NA	Not Above ...より上でない	CF=1 or ZF=1
G	Greater ...より大きい	ZF=0 and SF=OF
NLE	Not Less or Equal ...より小さくなく等しくない	ZF=0 and SF=OF
GE	Greater or Equal ...より大きいか等しい	SF=OF
NL	Not Less ...より小さくない	SF=OF
L	Less ...より小さい	SF ≠ OF
NGE	Not Greater or Equal ...より大きくなく等しくない	SF ≠ OF
LE	Less or Equal ...より小さいか等しい	ZF=1 or SF ≠ OF
NG	Not Greater ...より大きくない	ZF=1 or SF ≠ OF
C	Carry ...キャリがある	CF=1
NC	Not Carry ...キャリがない	CF=0
O	Overflow ...オーバフローがある	OF=1
NO	Not Overflow ...オーバフローがない	OF=0
S	Sign ...符号がある(負数)	SF=1
NS	Not Sign ...符号がない(非負数)	SF=0
P	Parity ...パリティがある	PF=1
PE	Parity Even ...パリティが偶数	PF=1
NP	Not Parity ...パリティがない	PF=0
PO	Parity Odd ...パリティが奇数	PF=0

〔表3〕LEA命令の動作

アドレスサイズ	オペランドサイズ	動作
16	16	sourceオペランドの実効アドレスを16ビットで計算し，destinationオペランドの16ビットレジスタにストアする
32	32	sourceオペランドの実効アドレスを32ビットで計算し，destinationオペランドの32ビットレジスタにストアする
16	32	sourceオペランドの実効アドレスを16ビットで計算し，ゼロ拡張した値を，destinationオペランドの32ビットレジスタにストアする
32	16	sourceオペランドの実効アドレスを32ビットで計算し，下位16ビットを，destinationオペランドの16ビットレジスタにストアする

ならCMPXCHG8B命令が使用できます．また，同じレジスタEDXのビット15が1ならCMOVcc命令が使用できることになります．

CPUID命令は，このほかにもプロセッサに関するいろいろな情報を提供してくれます．そのため，このCPUID命令は，回を改めてもう少し詳しく説明する予定です．

〔図6〕XLAT命令の動作

● LEA命令

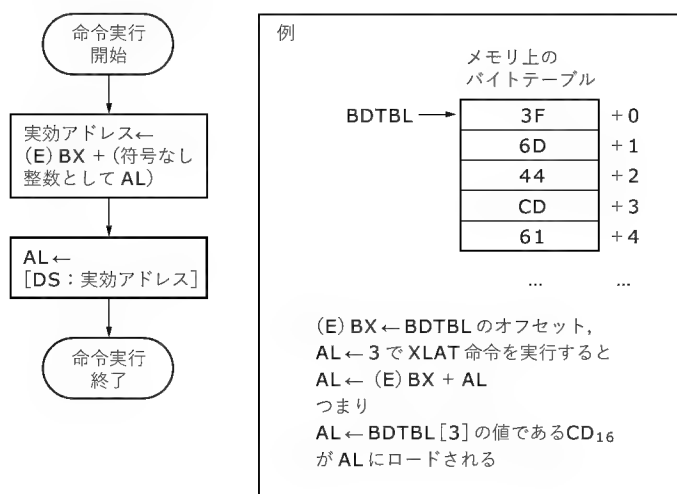
LEA命令は，「その他の命令」に分類されている命令で，x86系CPUすべてで使用できます．LEA命令は，転送元(SOU)で指定されたメモリ参照のオペランドで指定された実効アドレスを，転送先(DEST)のレジスタに転送します．

実効アドレスは，メモリアクセス時のオフセットとなる値です．i386以降の32ビットCPUでは，実効アドレスとして16ビットと32ビットのアドレスサイズが指定できます．

また，転送先のレジスタは，16ビットおよび32ビットのオペランドサイズの汎用レジスタが指定できます．

LEA命令はアドレスサイズとオペランドサイズの違いにより，**表3**のような動作をします．実際のMASMでの記述例を**リスト3**，gasでの記述例を**リスト4**に示します．

● XLAT命令

XLAT命令も「その他の命令」に分類されている命令で，x86系CPUすべてで使用できます．XLAT命令は，指定されたメモリ上のバイト値のテーブルから，インデックスで指定されたバイト値をレジスタALにロードするというものです(**図6**)．

〔リスト3〕**MASMのLEA，XLAT命令の記述例**

```
                              .586
                              .model flat

00000000                      .data

00000000 01                   dtByte  db      1
00000001 0002                 dtWord  dw      2
00000003 00000004             dtDWord dd      4

00000007  00000064 [          dtBA    db      100 dup(0)
           00
          ]

00000000                      .code
00000000  66| 8D 1D                   lea     bx,dtByte
          00000000 R
00000007  66| 8D 35                   lea     si,dtWord
          00000001 R
0000000E  66| 8D 3D                   lea     di,dtDWord
          00000003 R
00000015  8D 1D 00000000 R            lea     ebx,dtByte
0000001B  8D 35 00000001 R            lea     esi,dtWord
00000021  8D 3D 00000003 R            lea     edi,dtDWord

00000027  66| 8D 1C 35                lea     bx,dtBA[si+5]
          0000000C R
0000002F  66| 8D 9E                   lea     bx,dtBA[esi+5]
          0000000C R
00000036  8D 9E 0000000C R            lea     ebx,dtBA[esi+5]

0000003C  D7                          xlatb
0000003D  D7                          xlat
0000003E  D7                          xlat    dtBA

                              end
```

XLAT命令では，この3種類の記述が使用できる

〔リスト4〕**gasのLEA，XLAT命令の記述例**

```
 1                    .data
 2
 3  0000 01           dtByte:  .byte   1
 4  0001 0200         dtWord:  .word   2
 5  0003 04000000     dtDWord: .long   4
 6
 7  0007 00000000     dtBA:    .space  100,0
 7       00000000
 7       00000000
 7       00000000
 7       00000000
 8
 9                    .text
10  0000 668D1D00             leaw    dtByte,%bx
10       000000
11  0007 668D1D00             lea     dtByte,%bx
11       000000
12  000e 668D3501             leaw    dtWord,%si
12       000000
13  0015 668D3501             lea     dtWord,%si
13       000000
14  001c 668D3D03             leaw    dtDWord,%di
14       000000
15  0023 668D3D03             lea     dtDWord,%di
15       000000
16  002a 8D1D0000             leal    dtByte,%ebx
16       0000
17  0030 8D1D0000             lea     dtByte,%ebx
17       0000
18  0036 8D350100             leal    dtWord,%esi
18       0000
19  003c 8D350100             lea     dtWord,%esi
19       0000
20  0042 8D3D0300             leal    dtDWord,%edi
20       0000
21  0048 8D3D0300             lea     dtDWord,%edi
21       0000
22
23                   #        leaw    dtBA+5(%si),%bx
24                   #        lea     dtBA+5(%si),%bx
25  004e 668D9E0C             leaw    dtBA+5(%esi),%bx
25       000000
26  0055 668D9E0C             lea     dtBA+5(%esi),%bx
26       000000
27  005c 8D9E0C00             leal    dtBA+5(%esi),%ebx
27       0000
28  0062 8D9E0C00             lea     dtBA+5(%esi),%ebx
28       0000
29
30  0068 D7                   xlatb
31  0069 D7                   xlat
32  006a D7070000             xlatb   dtBA
32       00
33  006f D7070000             xlat    dtBA
33       00
```


メモリ上のバイト値のテーブルの先頭は，レジスタ(E)BXで指定します．この場合，CPUが16ビットでプログラムを実行している場合はレジスタBXとなり，32ビットでプログラムを実行している場合はレジスタEBXとなります．そのため，WindowsやLinuxの32ビットプログラムでは，レジスタEBXが使われます．

そして，インデックスはレジスタALで指定します．

このようにXLAT命令は，使用するレジスタを固定しているため，オペランドを必要としません．

ただし，アセンブラの記述上では，メモリ上のバイト値のテーブルの先頭をオペランドとして指定することも可能としています．この場合，指定されたオペランドはアセンブル時に，シンボルの検査のみが行われ，機械語命令の生成には使用されません．

実際のMASMでの記述例を**リスト3**，gasでの記述例を**リスト4**に示します．このリストを見るとわかるように，XLAT命令にオペランドを指定した場合，MASMでは正しく機械語命令を生成していますが，今回使用しているgasでは誤った機械語命令を生成しているのがわかります．そのため，gasでXLAT命令を使用する場合は，オペランドは指定しない方法で記述します．

*　　　　*

次回は，x86系の32ビットCPUで使用できる演算命令について説明します．

おおぬき・ひろゆき　大貫ソフトウェア設計事務所

組み込みシステム開発にデザインパターンを利用する

オブジェクト指向を使ったリアルタイム信号計測システムの開発

酒井由夫/松沢 航

はじめに

組み込みシステム機器におけるハードウェアとソフトウェアの比重は，年々ソフトウェア側に比重が移ってきています．その理由は，組み込みシステム機器に要求するユーザーの仕様が多種多様になってきており，それらの要望に答えるために大量のアプリケーションソフトウェアを作成して実装する必要が出てきたからだと考えられます．CPUの性能はどんどん上がっているので，そのようなさまざまな要求に対してもソフトウェアを実装することは物理的には可能ですが，それを短期間に実装しなければならないエンジニアの苦労は絶えません．

ソフトウェアに対する要求が増大し開発期間の短縮も実現しなければならないとなると，ソフトウェア技術者にかかる負担は増える一方です．このような，ソフトウェア技術者の八方ふさがり的な状況を打開するための一手段として，組み込みシステムへのオブジェクト指向設計が近年話題になっているのだと思います．

しかし，オブジェクト指向設計の一般的なモデルは，ビジネス系のアプリケーションソフトウェアが対象になっていることが多く，リアルタイム性が要求される組み込み機器で本当に使えるのかどうかがはっきりしない(雲をつかむような)と感じる方も少なくないでしょう．そのような中で，参考文献1)のようなeUMLによるオブジェクト指向組み込みシステム開発に関する本が出版されました．UMLを使って組み込みシステムの要求分析を行い，実装までもっていくという内容になっています．

しかし今回は，要求分析からオブジェクト指向設計を行うというよりは，「リアルタイム組み込みシステムでの信号計測」の例を題材に，組み込みシステムにおけるさまざまな制約条件下(ROM/RAM容量の制限，CPUのパフォーマンス，newや多重継承が使えないなど)で，オブジェクト指向設計をシステム全体のどの部分に利用するかに焦点を当て，オブジェクト指向設計におけるデザインパターンを使うことで解決できる問題がないかを考えていきたいと思います[注1]．

1. 解決すべき問題

リアルタイム信号計測系の組み込みシステムの実装では，信号計測のロジックが正しく動いているかどうかは，ハードウェアの試作が完成してから実際に信号をA-D変換してその動作を確認するという手順が一般的です．しかし，信号計測のソフトウェアロジックが複雑になればなるほど，実機を使ったデバッグには時間がかかり，ハードウェアができあがる前にロジックを検証したくなります．この場合，ソフトウェアロジックをパソコン上でシミュレーションし，その挙動をビジュアルに表示できるようにして動作を確認してから実機に実装するという形がとれれば，開発の期間を短縮し，ソフトウェアの信頼性を高めることができます．

このような実装前のシミュレーションは，一般的にはよく行われていると思いますが，シミュレーションで確認されたソースコードをできるだけそのままの形で実装したり，実装するソースコードとシミュレーションで使用するソースコードの大部分が共通であり，かつ，常に両方とも最新の状態を保つことができれば，問題が起きたときの検証作業や，新しいロジックを一時的に試してみることが容易になります．

また，信号計測系のソフトウェアでは，取り込んだ入力信号がどのように機器の中で二次処理されたのかをビジュアルに確認することが必要になります．このとき，シミュレーション環境としてパソコン上でVisual C++やC++ Builderといったツールを使うことができれば，実機上でデバッグ用のGUIソフトウェアを自作するよりも圧倒的に早く，かつ豊かなユーザーインターフェースを使ってデバッグ環境を構築できます．

さらに，組み込みシステムに固有の部分やシミュレーション環境に固有の部分，また信号計測の心臓部を切り分けることにより，将来発生する可能性のある突然のハードウェアの変更や信号計測部への改善要求の実施をスムースに行うことができます．

これは，過去のソフトウェア資産をブラックボックスとして

注1：本稿は，これまでC++を使ってオブジェクト指向設計をリアルタイム組み込みシステムに導入したことのないプロジェクトチームをおもな読者対象として考えている．

コラム1
デザインパターンとは？

バークレーの名誉教授であり，かつ建築家でもあるC.Alexanderは1960～1970年代に「パターンとはわれわれの身のまわりで何回も起こる問題とそれに対する解決のポイント」であるとし，建物や町の設計，オフィスのデザインやレイアウトには，必ずオーソドックスなパターンが存在することを彼の著書*A Pattern Language*（邦訳：『パタン・ランゲージ』）と*The Timeless Way of Building*（邦訳：『時を超えた建設の道』）で，建築に関するパターンとして示しています．

参考文献3）の著者であるErich Gammaら4人（通称GoF：Gang of Four）は，オブジェクト指向ソフトウェア設計において，私たちが何回も遭遇する問題とそれに対する解決方法を23の具体的なパターンとしてまとめました．これらのソフトウェア・デザインパターンの応用は，いわゆる「車輪の再発明」[注A]を防ぎ，大規模なソフトウェアに対し，これらのパターンを組み合わせることにより，より少ないエネルギで確実にソフトウェア設計を行うことを可能にしました．これは，囲碁や将棋の「定石」に似ています．

GoFらの示したオブジェクト指向言語におけるこれらのパターンの本質は，「オブジェクトの生成」，「データ構造」，「ふるまいの変更」の三つに集約されますが，組み込みシステムのプログラミングにおいては，これらの本質すべてをフル活用することはリソースや開発環境の問題からも難しいといえます．

筆者らが今回紹介したのは「ふるまいの変更」に対して開発工数を減らし，ロバスト[注B]を大きくするための「Template Method パターン」や「Strategy パターン」の応用事例です．これらのパターンの本質は，基底クラスにおいて定義したインターフェースや呼び出しの手順を，派生クラス側でオーバライドすることによって，変更の必要な箇所のみを差し替えることにより，安全にソフトウェアの機能拡張や修正などの開発が可能な点にあるといえます．このように関数をオーバライドすることにより機能を差し替える手法をフックメソッドと呼びます．

注A：「車輪の再発明」とは，多くのプログラマがすでに誰かによって目的とする機能が実装されたモジュールが存在するにも関わらず，それを利用するのを避け，新たに自分で同じ機能のモジュールを再開発することの比喩．

注B：ここでの「ロバスト」とは，要求仕様の変更に対してどのくらいの許容度があるかという意味．

手を入れないのではなく，変更が予想される部分やハードウェアに依存する部分を上手に切り分け，積極的かつ安全にアップグレードを行っていくという考え方に基づいています．ユーザーからの要求が多種多様でその仕様が短期間で変化している現状では，変更のあった要求を実現している部分（モジュール）を素早くかつ安全にアップグレードしたいものです．担当者に「その部分を変えるとシステム全体にどのような影響を及ぼすのかわからないので変更できません」とは言わせたくありません．

この提案は，オブジェクト指向設計をユーザーの要求分析から行うトップダウンの手法とは異なりますが，組み込みシステムの厳しい制約条件を十分に理解しつつ，オブジェクト指向設計のメリットを生かすという意味では，有効な手段であると考えられます．

また，オブジェクト指向設計自体も，本来何を「目的」ととらえるかによっては，それに使用されるデザインパターンも変わってくるし，解決できる方法は一つではないという特徴をもっていると思います．オブジェクト指向設計を大上段に構えるのではなく，対象のシステムに使えるデザインパターンを上手に利用するという観点からとらえた例と考えてください．

2. リアルタイム信号計測システムの例

まず，本稿で解説するリアルタイム信号計測システムの例について，目的や性能を説明します．

● 取り扱う入力信号

連続的に変化する50Hz以下の信号です．

● システムの目的

10Hz以上の周波数成分を除去し，1Hz以下の波形は減衰させないようようなフィルタを設計し実装します．具体的には，入力信号である1Hzの正弦波に，ノイズ成分に見立てた10Hzの正弦波を重畳させ，設計したフィルタにより10Hzのノイズが確実に除去されるかどうかを確認します（2種類のフィルタを設計し，より効果の高いほうを採用する）．

● システムの機能および性能

▶パソコン上の仮想（バーチャル）システムでの機能および性能

① 仮想システムのプラットホームはDOS/Vパソコンを想定する
② ファイルから検討用に作成したダミーの入力信号（1Hzの正弦波に10Hzの正弦波を重畳させたもの）など，任意の信号データファイルを読み込むことができる
③ 入力信号をウィンドウに表示できる
④ 2種類のハイカットフィルタ（カットオフ周波数：8.4Hz，4.3Hz）をON/OFFできる
⑤ フィルタをかける前の波形と，フィルタをかけた後の波形をウィンドウに表示できる
⑥ 時間軸のスケールを変更できる
⑦ フィルタによる波形の遅れ時間を補正できる
⑧ デモ用の波形（5Hzの正弦波）でフィルタの効き目を試すことができる

▶実（リアル）システムでの機能および性能

① バーチャルシステムが入力信号をファイルから読み込むのに対して，リアルシステムでは入力信号をA-D変換で取り込む
② リアルタイムOSに対応している（リアルタイムOSがなくて

も実装は可能)

③ リアルシステムとバーチャルシステムのシステムの違いに依存する部分(信号の入力や，OS)以外はシステム間で共通に利用できる

*　　　　　　　　*

システムの全体図を**図1**に，実システムと仮想システムの関係を**図2**に，デモ波形の加工例を**図3**に示します.

3. 組み込みシステムの制約条件

今回の信号計測システムを実装するCPUは，日立製作所製のH8Sを想定しています．H8Sシリーズは，内部32ビット構成のCPUをコアに必要な周辺機能を集積した高性能のCISCマイクロコンピュータです.

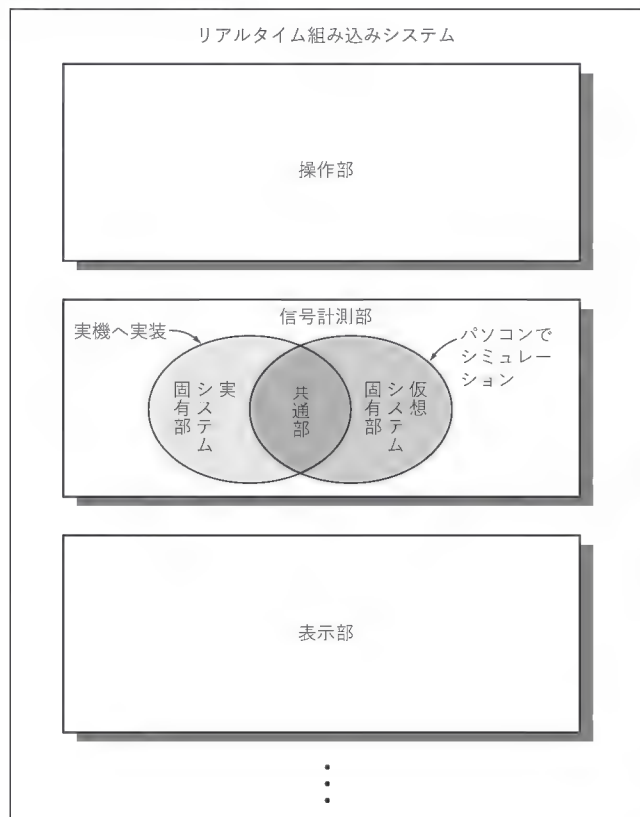
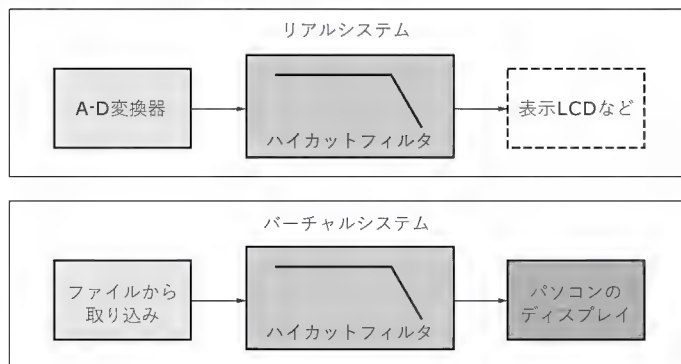
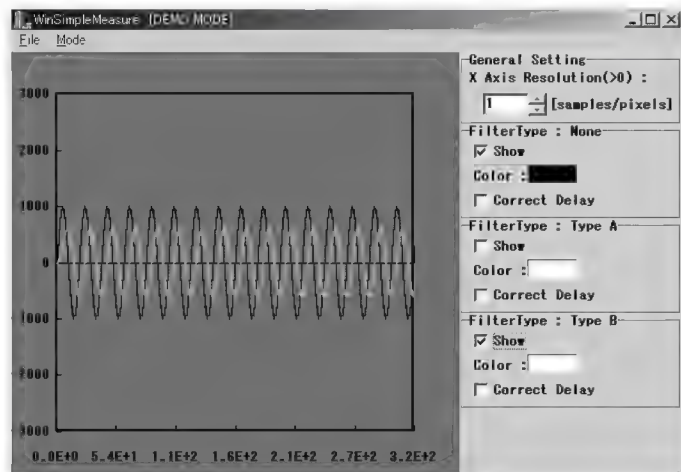
日立製のマイコンで特筆すべき点は，日立純正のコンパイラが標準でCとC++のソースを両方とも扱えるという点です．パソコン上ではC++のソースがコンパイルできることは珍しいことではありませんが，組み込みの世界でかつCISCマイコンのコンパイラがC++に対応しているという点は貴重です．しかし，H8またはH8Sシリーズに用意されているC/C++コンパイラでは，次のような制限事項もあります.

- 多重継承は使えない
- 次のような低水準インターフェースルーチンはコンパイラの標準ライブラリには含まれておらず，ユーザーが作成しなければならない

　`open`　：ファイルのオープン
　`close`　：ファイルのクローズ
　`read`　：ファイルからの読み込み
　`write`　：ファイルへの書き出し
　`lseek`　：ファイルの読み込み/書き出しの位置の設定
　`sbrk`　：メモリ領域の確保

この中でもっともつらいのは，「メモリ領域の確保」をユーザーが用意しなければならない点です．メモリ領域をユーザーが確保するということは，すなわち，メモリを動的に確保・開放(`new`や`delete`)する操作を行ったときに，CPUが管理しているメモリ資源から，要求された領域を確保したり，開放する作業をユーザー自身が考えて実装しなければいけないということです.

RAMは組み込みシステムの限られた資源の一つですから，このことを知らないプログラマが仮想システム上で`new`を連発し

〔図1〕システムの全体図

〔図2〕実システムと仮想システムの関係

〔図3〕デモ波形の加工例

たり，大きな領域を動的に確保するプログラムを書くと，実システム上でエラーになるなど，システムが破綻する可能性があります．もっとも悪いのは，メモリの確保と開放を繰り返して，空きのメモリ領域が分散してしまったり，メモリ確保のタイミングによってヒープ領域がオーバしてしまうような場合です．このような場合は，不具合の状況の再現が困難な場合が多く，不具合発生の手順が一定しないこともあります．メモリ領域の確保の低水準インターフェースルーチンを自作し，気を付けてnewを使うよりは，いっそnewを使用しないで，プログラミングする方法をとったほうが安全であると考えます．

4. システムのクラス図の説明

クラス図を**図4**(次頁)に示し，以下でクラスの説明をします[注2]．

- CMesureSignal クラス：信号計測のメインクラス

このクラスの公開されたサービス(publicなメンバ関数)を呼ぶことで，本信号計測システムを実現します．

- CBehaviorOnSystem：システムに依存する基底クラス

リアルシステムおよびバーチャルシステムに依存する部分を集めて，それらのシステム間の違いをこの基底(抽象)クラスを介して共通化します．

- CBehaviorOnRealSystem：システムに依存する派生クラス(実システム)

CBehaviorOnSystemの派生クラスです．実システム固有の処理が記述されています．仮想システムの処理(メンバ関数)とインターフェースが統一化されています．波形データをA-D変換器から得ます．

- CBehaviorOnVirtualSystem：システムに依存する派生クラス(仮想システム)

CBehaviorOnSystemの派生クラスです．仮想システム固有の処理が記述されています．実システムの処理(メンバ関数)とインターフェースが統一化されています．波形データをファイルから得ます．

- CBehaviorOnVirtualDemo：システムに依存する派生クラス(デモシステム)

CBehaviorOnSystemの派生クラスです．仮想デモシステム固有の処理が記述されています．実システムの処理(メンバ関数)とインターフェースが統一されています．波形データをあらかじめ用意したデモデータから得ます．

- CHighCutFilter：ハイカットフィルタの基底クラス

さまざまなハイカットフィルタを実装するための基底(抽象)クラスです．このクラスを介して，2種類の異なるハイカットフィルタのクラスを派生させます．

- CHighCutFilterTypeA：ハイカットフィルタの派生クラス
  CHighCutFilterの派生クラスです．カットオフ周波数8.4HzのFIRフィルタです．
- CHighCutFilterTypeB：ハイカットフィルタの派生クラス
  CHighCutFilterの派生クラスです．カットオフ周波数4.3HzのFIRフィルタです．
- 設計のポイント
- フィルタのクラスはスケルトンの基底クラス(抽象クラス)に対し，いくつかの派生クラスを作成し，検討する二つのハイカットフィルタを派生クラスに実装する(Template Methodパターン)
- 実システムまたは仮想システムに依存する機能を，同様にスケルトンの基底クラス(抽象クラス)に対し，実システムと仮想システムの派生クラスを作成して実装する(Template Methodパターン)
- 実システムまたは仮想システムに依存する部分と，ハイカットフィルタのアルゴリズムをカプセル化し，それを利用するクライアント(CMeasureSignalクラス)から独立させる(Strategyパターン)

5. CMesureSignal クラス：信号計測のメインクラスの利用方法

信号計測システムを利用するユーザーは，CMesureSignalクラスの公開されたサービス(publicなメンバ関数)を呼ぶことによって，CMesureSignalに用意された機能を実現します．

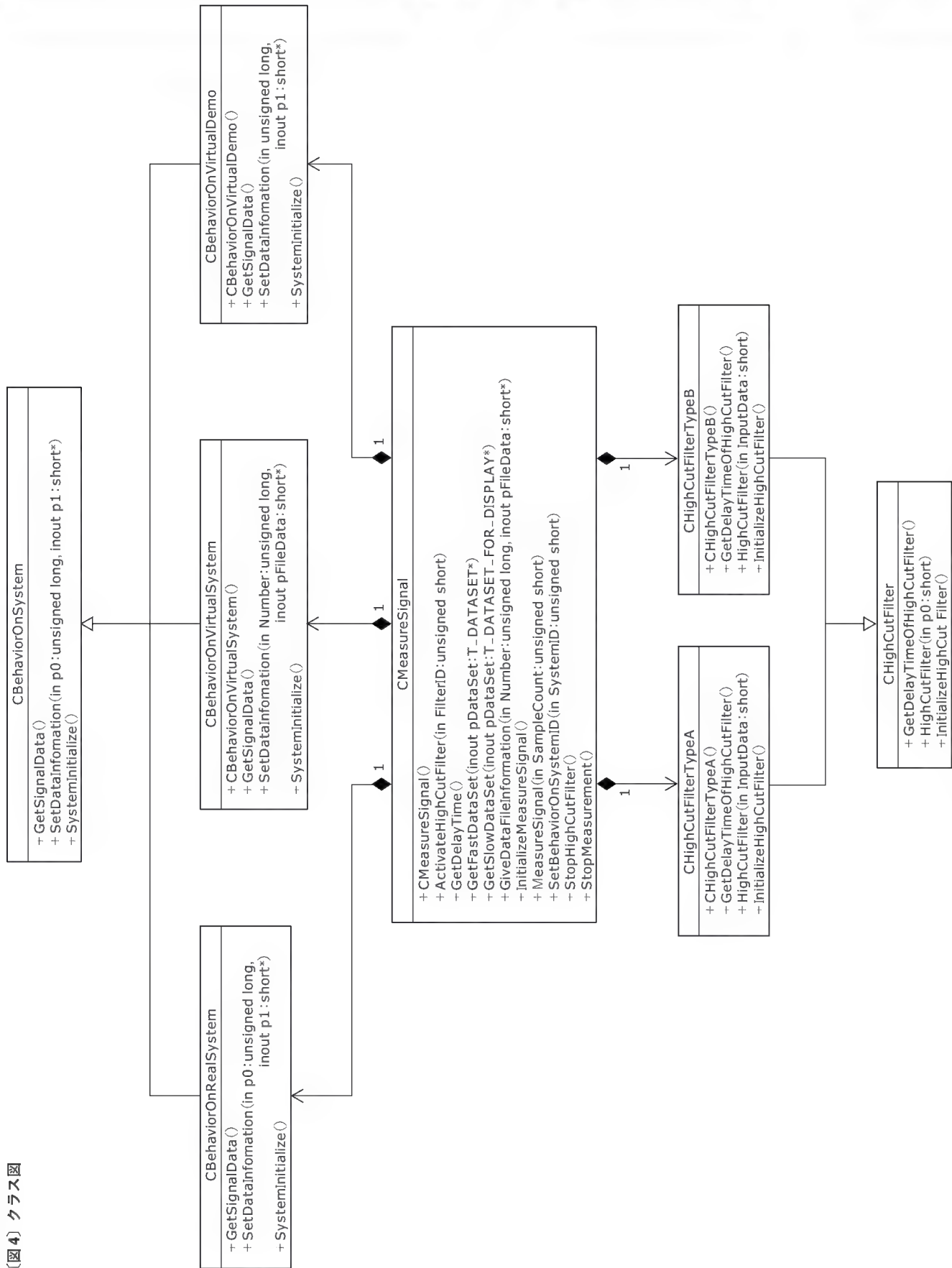
ユーザーが利用可能なサービス(メンバ関数)を**リスト1**に示します[注3]．

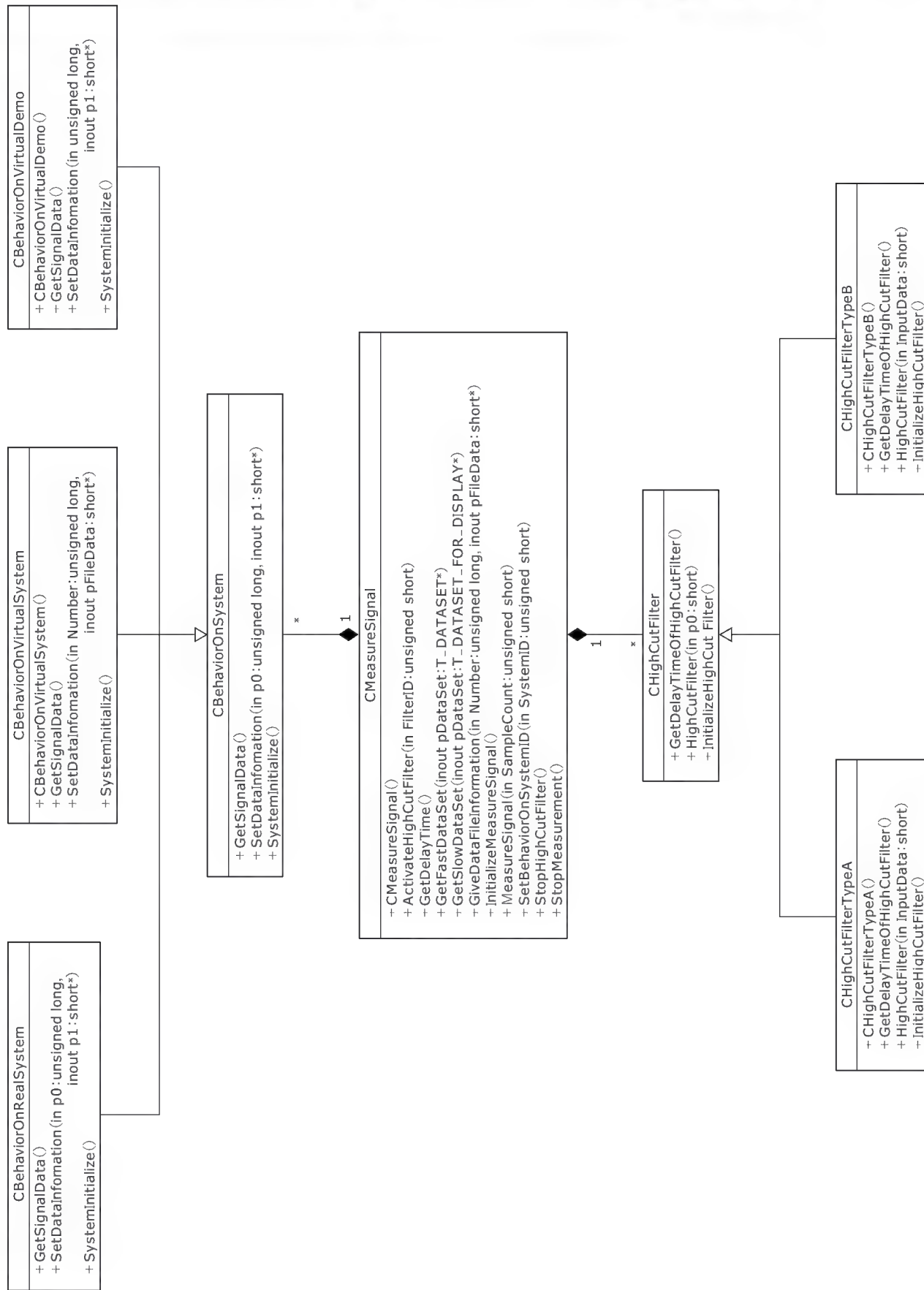
〔リスト1〕ユーザーが利用可能なサービス(メンバ関数)

```
// 信号計測メイン
void MeasureSignal( unsigned short SampleCount );
// ハイカットフィルタを有効にする
void ActivateHighCutFilter( unsigned short FilterID );
// ハイカットフィルタを無効にする
void StopHighCutFilter( void );
// 信号の計測を止める
void StopMeasurement( void );
// データファイル情報を与える
void GiveDataFileInformation( unsigned long Number,
signed short* pFileData );
// 遅れ時間を取得する(単位:データ数)
unsigned short GetDelayTime( void );
// 早い信号データ引き渡し関数(瞬時データ取得用)
void GetFastDataSet( T DATASET *pDataSet );
// 遅い信号データ引き渡し関数(表示用データ取得用)
void GetSlowDataSet( T DATASET FOR DISPLAY *pDataSet );
// 信号計測初期化関数
void InitializeMeasureSignal( void );
// システムを設定する
void SetBehaviorOnSystem( unsigned short SystemID );
```

注2：ここで紹介したクラスは信号計測に関連するクラスのみで，表示やファイル読み込みに関するクラスは解説していない．

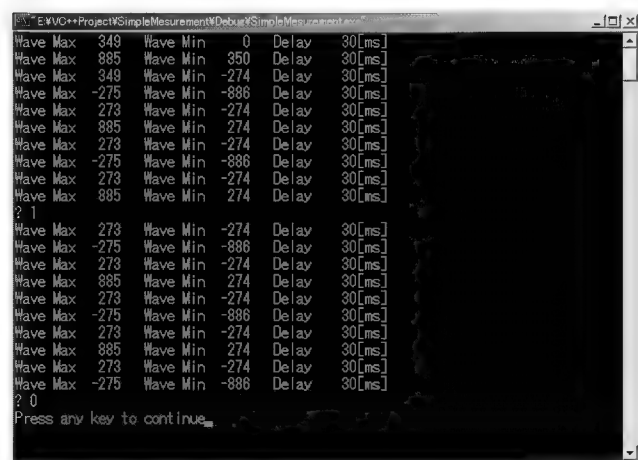
注3：CMesureSignalクラスを利用するユーザーがGiveDataFileInformation関数を呼ぶことにより，仮想システムで使用する入力信号データが格納されている配列のポインタとデータ数をCMesureSignalクラスに知らせる．実機に実装したときにこのサービスを呼んでも何も起こらない(実システムではデータをA-D変換器から取得するため)．

〔図4〕クラス図

(a) 組み込み系の new を使わない場合のクラス図

〔図4〕クラス図（つづき）

（b）パソコン系の new を使った場合のクラス図

〔リスト2〕Visual C++のコンソールアプリケーション(GUIなし)で実施した例

```
int main(void)
{
    // インスタンスの生成
    CMeasureSignal *pMeasureSignal = new CMeasureSignal;

    // 表示用データの受け渡し変数の生成
    T_DATASET_FOR_DISPLAY   WaveDataSet;

    // システムを設定する
    //--------------------------------------
    //      0 : バーチャルシステム
    //      その他 : デモシステム
    //--------------------------------------
    pMeasureSignal->SetBehaviorOnSystem( 1 );

    // ハイカットフィルタを有効にする
    //--------------------------------------
    //      1 : カットオフ周波数 8.4Hz
    //      2 : カットオフ周波数 4.3Hz
    //   その他 : 未定義
    //--------------------------------------
    pMeasureSignal->ActivateHighCutFilter( 1 );

    do {
         for ( int i = 0; i < 10; i++ ) {

              // 計測を開始する
              pMeasureSignal->MeasureSignal( 5 );

              // 表示用の波形データを得る
              pMeasureSignal->GetSlowDataSet( &WaveDataSet );

              // 遅れ時間を得る
              int DelayTime = pMeasureSignal->GetDelayTime()
                                                          * 10;

              // 各種データを表示する
              printf( "Wave Max%6d   Wave Min%6d
                                             Delay%6d[ms]¥n"
                         , WaveDataSet.max
                         , WaveDataSet.min
                         , DelayTime
                         );
         }

         // ループ終了か?
         short temp;
         cout << "? ";
         cin  >> temp;
         if ( temp == 0 ) break;

    } while ( 1 );

    // 波形の計測を止める
    pMeasureSignal->StopMeasurement();

    return 0;
}
```

〔図5〕コンソールアプリケーションでの実施例

6. 仮想システムにおけるCMesureSignalクラスのもっとも単純な利用例(仮想システムの実施例1)
――Visual C++のコンソールアプリケーション(GUIなし)で実施した例

リスト2では，仮想システムにおいて，CBehaviorOnVirtualDemoクラスにあらかじめ用意された5Hz，sin波デモ波形データを用いて8.4Hzのハイカットフィルタを通した結果を，5サンプリング(50ms)ごとに取得し，波形描画に利用する最大データおよび最小データ，フィルタの遅れ時間を10行ごとにテキストで表示しています．

CMesureSignalクラスで扱う入力信号のサンプリング時間は10msで固定としています[注4]．MesureSignal()メンバ関数の引き数は，フィルタリング処理を何回連続して行い結果を得るかを指定します．

図5では，50ms(5サンプリング)ごとにフィルタリング処理を行い，その結果得られた波形の最大値，最小値を得ています．MesureSignal関数を利用するユーザーはこの最大値，最小値を加工し垂直な線を引くことで，フィルタ通過後の波形を描画することができます．

また，仮想システム(パソコン)での描画において時間軸のスケールを変更したいときは，MesureSignal関数の引き数を変更することにより描画するためのデータを得ることができます．

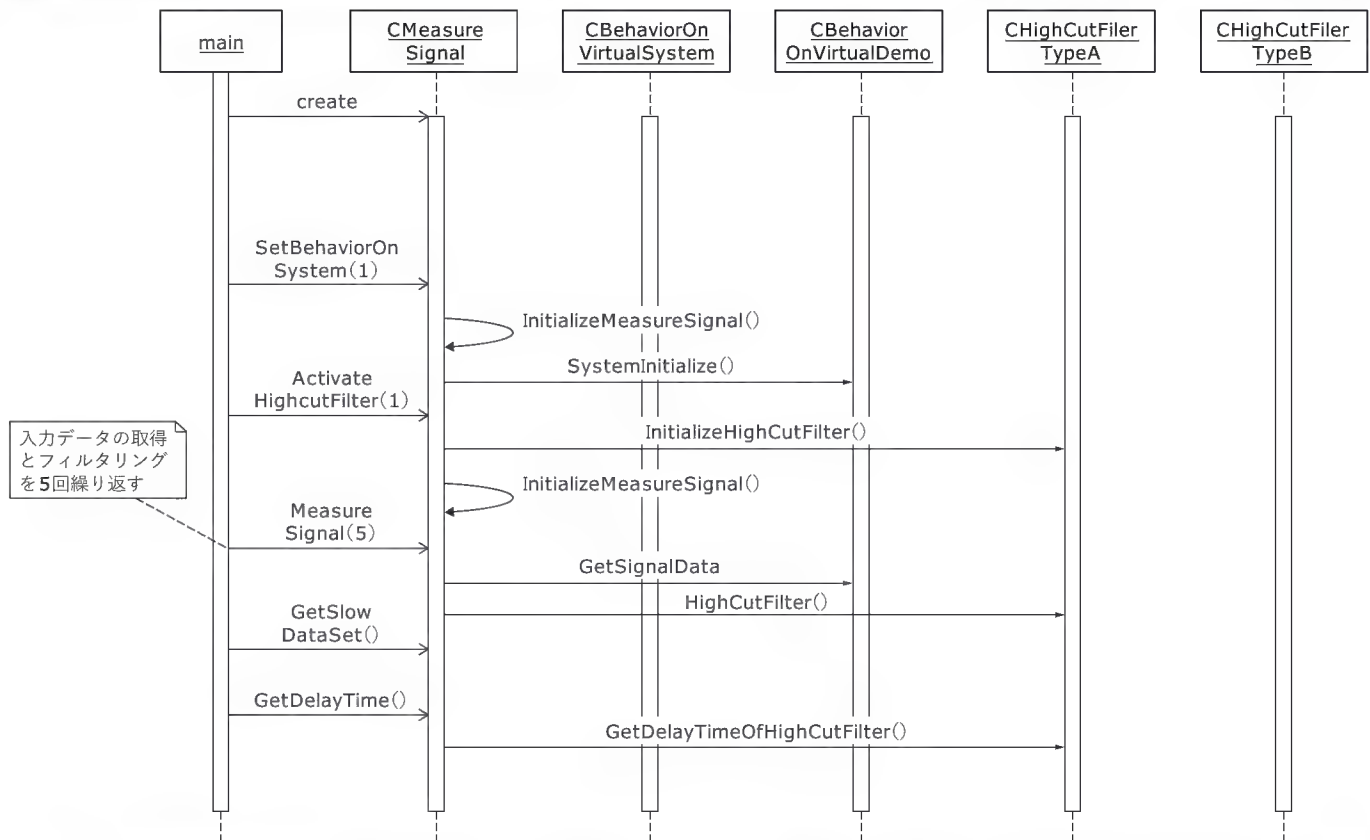
シーケンス図を**図6**に示します．

7. 仮想システムの実施例2
――C++ Builderを使ったアプリケーションプログラムの例

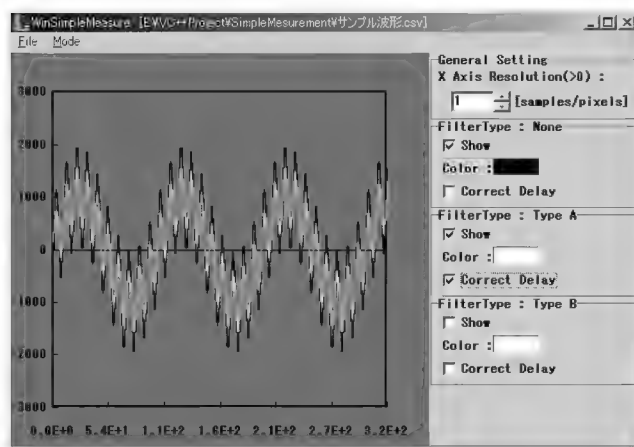
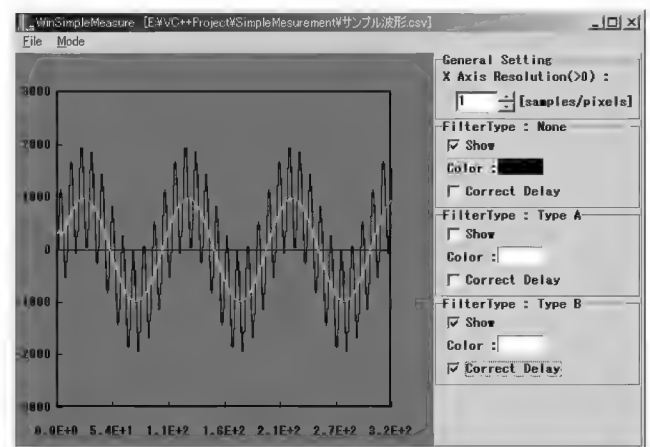
C++ Builderを使ったアプリケーションの例です．信号処理の外側からはCMesureSignalクラスに用意されたサービス(publicなメンバ関数)をアクセスすることによって，計測処理を実現しています．

図7の例では，1Hzの正弦波に10Hzの正弦波を重畳されたサンプル波形をファイルから読み込み，8.4Hzのハイカットフィルタと，4.3Hzのハイカットフィルタを通すことで，1Hzの正弦波に重畳した10Hzの正弦波がどれくらい取り除けるのかを検証しています．**図7(a)**では，カットオフ周波数8.4Hzのフィルタでは

注4：サンプリングタイムを変更することは，システム全体(とくにフィルタの差分方程式に対して)の修正が必要となってしまうので，できない．サンプリング間隔は将来も変わらないという前提のシステムであると考えてほしい．

〔図6〕シーケンス図

〔図7〕サンプル波形の加工例

(a)

(b)

10Hzの正弦波は十分に取り除かれておらず，**図7**(**b**) 4.3Hzのフィルタでは完全に除去できたことがわかります．この実施例から，10Hzのノイズを取り除くためには，カットオフ周波数4.3Hzのハイカットフィルタのほうが適していることがわかりました．

ファイルを選択しているようすを**図8**に示します[注5]．

8. 実システムへの実装

仮想システムにおいて，CMesureSignalクラスの動作とハイカットフィルタの選択が完了しました．CMesureSignalク

注5：元波形の描画，フィルタ通過後の波形の描画，波形色の選択，ファイルからのデータ読み込み，フィルタによる遅れ時間の補正，デモ波形の表示などは，C++ Builderで用意されたクラスライブラリを使って実現している．

〔図8〕ファイルを選択している図

〔リスト3〕リアルシステムにおける **CMeasureSignal** クラスの宣言

```
class CMeasureSignal
{
public:
  :
  :

protected:

  :
  :

    //**********************************************
    // 機器固有オブジェクト生成
    //**********************************************
    //----------------------------------------------
    // 実システムからの信号計測オブジェクトを生成
    CBehaviorOnRealSystem mObjectMeasOnSystem;   ← コメントアウトを解除
    //----------------------------------------------
    // 仮想システムからの信号計測オブジェクトを生成
//  CBehaviorOnVirtualSystem mObjectMeasOnSystem;   ← コメントアウト
    //----------------------------------------------
    // 仮想システム上での信号計測デモオブジェクトを生成
//  CBehaviorOnVirtualDemo mObjectMeasOnDemo;   ← コメントアウト

    //**********************************************
    // ハイカットフィルタオブジェクト生成
    //**********************************************
    //----------------------------------------------
    // ハイカットフィルタAオブジェクトを生成
//  CHighCutFilterTypeA mObjectHighCutFilterA;   ← コメントアウト
    // ハイカットフィルタBオブジェクトを生成

    CHighCutFilterTypeB mObjectHighCutFilterB;
};
```

ラスを実装するために，CBehaviorOnVirtualSystem クラスと CBehaviorOnVirtualDemo クラスを切り離して(コメントアウトして)，CBehaviorOnRealSystem クラスを実装します(**リスト3**．条件コンパイルでも可)．

仮想システムから実システムへの移行について，**図9**に示します．

● 実装のためのポイント

① new を使わない

仮想システム(パソコン)では new を使ってもコンパイルエラーにはなりませんが，実システム上では，ユーザーがメモリの確保，解放の低水準インターフェースを作成しないと実装できないので，new は使わないようにします．

〔リスト4〕**new** を使わない

```
     :
CBehaviorOnVirtualSystem  mObjectMeasOnSystem;  ← インスタンスを名前を付けて生成する
     :
CBehaviorOnSystem        *mpMeasOnSystem;  ← クラスオブジェクトのポインタ変数を作る
     :
mpMeasOnSystem           = &mObjectMeasOnSystem;  ← オブジェクトポインタを代入する
     :
```

〔リスト5〕**CBehaviorOnSystem**(基底)クラスの定義

```
class CBehaviorOnSystem
{
public:
    // 入力仮想関数
    virtual signed short GetSignalData( void ) = 0;  ← 純粋仮想関数
    // データ情報を設定する
    virtual void SetDataInfomation( unsigned long Number, signed
         short* SignalData ) = 0;  ← 純粋仮想関数
};
```

〔リスト6〕**CBehaviorOnRealSystem**(派生)クラスの定義

```
class CBehaviorOnRealSystem : public CBehaviorOnSystem
{
public:
    // A-D変換器からデータを取り込む関数
    virtual signed short GetSignalData( void );  ← 純粋仮想関数を再定義する
    // データ情報を設定する
    virtual void SetDataInfomation( unsigned long Number, signed
         short* SignalData ){}  ← リアルタイムシステムでは使用しないので空の関数として再定義する
```

このため，面倒でも名前をつけてインスタンスを生成し，コンストラクタの中などで，そのクラスオブジェクトのポインタをメンバ変数にしたクラスのポインタ変数に代入します(**リスト4**)．

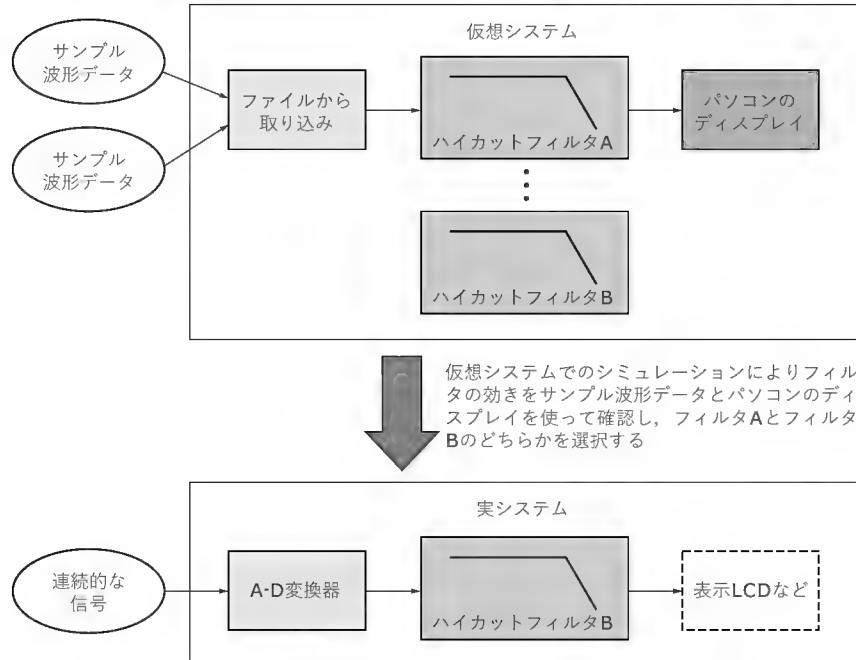
② 実システムおよび仮想システムに固有の機能は全体の機能から切り離してクラスにまとめる

CBehaviorOnSystem クラス(**リスト5**)を基底クラスにして，実システムと仮想システムに固有の機能(たとえば入力信号の取り込みなど)を純粋仮想関数としてすべて定義し，実システムに固有な機能は CBehaviorOnRealSystem クラス(**リスト6**)で，仮想システムに固有な機能は CBehaviorOnVirtualSystem クラス(**リスト7**)および CBehaviorOnVirtualDemo クラスで基底クラスの純粋仮想関数を派生クラスの中で再定義して実装します[注6]．

CBehaviorOnSystem クラスで定義した純粋仮想関数の使用状況について**表1**に示します．

③ CBehaviorOnRealSystem クラスの信号入力関数“GetSignalData()”(**リスト8**)の中で，サンプリングのタ

注6：実システムのクラスと仮想システムのクラスのインターフェース(メンバ関数)は一致していないと，基底クラスの派生にできないので，それぞれクラスに関係のないメンバ関数も純粋仮想関数として基底クラスに登録しておき，派生クラスの中で使用しない純粋仮想関数は，再定義する際に中身を空にしておく．一見その作業は無駄なように見えるが，実システムに固有な機能のみを切り離し，それ以外の部分をパソコンの仮想システム上で検証できるので，このクラス派生のしくみはとても有効である．実機にソフトウェアを実装する際，実システム以外の部分はパソコン環境で検証済みであり，かつ，ロジック上の不具合が発見された場合にも，仮想システム上で Visual C++ や C++ Builder のデバッグ機能を使って確認できるメリットがある．

〔図9〕
仮想システムから実システムへの移行

〔リスト7〕CBehaviorOnVirtualSystemクラスの定義

```
class CBehaviorOnVirtualSystem : public CBehaviorOnSystem
{
public:
    // コンストラクタ
    CBehaviorOnVirtualSystem( void );
    // 仮想入力からデータを取り込む関数
    virtual signed short GetSignalData( void );  ← 純粋仮想関数を再定義する
    // データ情報を設定する
    virtual void SetDataInfomation( unsigned long Number,
         signed short* SignalData );  ← 純粋仮想関数を再定義する
protected:
    :
    :
};
```

〔リスト8〕実システムのGetSignalData()関数

```
short CBehaviorOnRealSystem::GetSignalData()
{
    signed short DataValue = 0;

    // ここで10msのタイミングを作る
    // たとえば，以下のようにリアルタイムOSのシステムコールを呼んで一定時間待ち
    // 状態に移行する
    // wai tsk( 10 );

    // ここでA-D変換器からのデータを取得する
    // たとえば以下のような関数からデータを取得する
    // DataValue = GetADData();

    return DataValue;
}
```

イミングを作る

GetSignalData()関数の中で，リアルタイムOSの時間待ちタスクを呼び，サンプリング時間である10msのタイムベースを作り，その後，A-D変換器で取得したデータを読み込むようにします注7.

〔表1〕CBehaviorOnSystemクラスで定義した純粋仮想関数の使用状況

純粋仮想関数	実システム	仮想システム	仮想デモシステム
GetSignalData()	使用する	使用する	使用する
SetDataInfomation()	使用しない	使用する	使用しない

〔リスト9〕
CMeasureSignalクラスのインスタンスにイベントを伝えたいとき

```
static CMeasureSignal MeasureSignal;  ← スタティックにインスタンスを生成し利用する
    :
    :

int main(void)
{
    :
    :
    // ハイカットフィルタを有効にする
    MeasureSignal.ActivateHighCutFilter( 1 );  ← 直接サービスを呼ぶ場合
    :
    :
}
        スタティックに生成したインスタンスのメンバを呼ぶダミーの関数を用意する
void ActivateHighCutFilter( unsigned short FilterID )
{
    MeasureSignal.ActivateHighCutFilter( FilterID );  ← Cのグローバルな関数を介して呼ぶ場合
}
```

④ CMeasureSignalクラスのインスタンスにイベントを伝えたいとき

CMeasureSignalクラスのインスタンスに外側からイベントが発生したことを知らせたいとき(たとえば実機システム上の何らかのキーが押されて割り込みイベントが発生したときなど)が

注7：サンプリングタイミングの作り方や，A-D変換の方法は，それぞれのシステムで独自の方法を選択してほしい(リアルタイムOSは必須条件ではない)．大事なことは，純粋仮想関数であるGetSignalData()を再定義し，その中でサンプリングタイミングを作り，入力データを得る処理を行う，または，それらの処理を行う関数を呼ぶことである．

あります(**リスト9**，前頁).

このような場合に，CMeasureSignal クラスの公開されたメンバ関数を呼ぶためには，イベントの発生を受けた関数(C の関数かもしれない)が CMeasureSignal クラスのインスタンスのポインタを知らなければ，公開されたサービスを呼ぶことはできません．

しかし，サービスを呼ぼうとしている関数が C の関数で C コンパイラでコンパイルするような場合は，C++ のオブジェクトポインタを受けることはできません．

このような場合は，クラスのインスタンスをスタティックに生成し，生成したインスタンスのメンバを呼ぶためのグローバルなダミーの C の関数を用意して，それらの関数を通して，インスタンスのメンバを呼ぶようにします．こうしておけば，純粋な C の関数から C++ のクラスメンバを呼び出すことが可能になります(ラッピング).

9. 考察

● フィルタの再検討について

今回の例では 4.3Hz のハイカットフィルタを採用しましたが，システムに対する要求が変化し必要とされる信号の周波数成分が変化したら，CHighCutFilterTypeC や CHighCutFilterTypeD といった検討用の派生クラスを作って，仮想システム上で動作を確認し実装します．ハイカットフィルタのインターフェースが同じであれば，派生クラスのみを追加すればよいので，新しいフィルタを安心して試すことができます．CMeasureSignal クラスを使用するユーザーがフィルタが変わったことで考慮しなければいけないのは，ハイカットフィルタの遅れ時間だけです．なぜなら，フィルタを再設計すると，フィルタの遅れ時間はたいてい変わってしまうからです．この遅れ時間を得

コラム2
組み込みシステムへオブジェクト指向設計を導入することについて

組み込みシステムにおいて，これまで C 言語でシステムを構築してきたプロジェクトチームがオブジェクト指向設計を導入するかどうかは，悩むところだと思います．その判断材料のために，今回の信号計測への導入例に絞ってメリット，デメリットをあげてみました．

● メリット

① システムの重要なソフトウェア機能について，積極的かつ安全に新しい機能入れ替えを行うことができる
② 実機での動作をパソコンによるシミュレーション環境で確認できる
③ C++ を使うことで，機能モジュールのクラスによる切り分けや，外部に公開したくない変数や関数の隠蔽，外部とのインターフェースの明確化が可能になる

● デメリット

① CPU のパフォーマンスおよび使用するメモリ資源が，わずかながら C 言語で書かれたシステムより増加する
② 実績のある C 言語で作成した資産がある場合，一部再設計する必要がある
③ オブジェクト指向設計(とくに C++ 言語)を学習するのに時間がかかる
④ オブジェクト指向設計(新しい試み)を導入する際の許可またはプロジェクトメンバからの賛同を得ることが難しい

*　　　　*

このようなメリット，デメリットを秤にかけて，デメリットのほうが大きいと感じる場合は，オブジェクト指向設計の導入を慎重に考えたほうがよいかもしれません．そのようなときは，今回の例のようにシステムの一部についてオブジェクト指向設計を実施してみるのがよいでしょう．

また，デメリット③の「C++ 言語を学習するのに時間がかかる」は現実的には大きな問題です．たいていは誰でも，新しい技術を身につけることを嫌がるからです．しかし，時代の流れはオブジェクト指向設計に大きく傾いているので，その技術を習得しやすい環境はすでにあります．次のような手順で学習すれば，比較的短期間にオブジェクト指向プログラミングの技術を身につけることができるでしょう．

● オブジェクト指向プログラミングの学習手順

① Visual C++ や C++ Builder の入門書，付属のチュートリアルを使って，まず実際に自分で C++ のプログラムを書いてみる
② C++ の基本概念をきちんと理解する(参考書『C++ プログラミングスタイル』など)
③ C++ 特有の機能や制限を学習する(参考書『改訂 C++ のからくり』など)
④ ターゲットシステムで使用する CPU の C++ コンパイラの制限事項をよく読む(参考書：コンパイラマニュアル)
⑤ オブジェクト指向設計におけるデザインパターンを学び，ターゲットシステムのどの部分に利用するかを考える(参考書『オブジェクト指向設計におけるデザインパターン』など)

*　　　　*

このように学習のパターンを作り，日程をスケジューリングしてしまえば，短期間で C++ の基本を学ぶことが可能だと思います．オブジェクト指向プログラミングの学習期間を設定せずに，直接設計を始めるのは危険です．なぜなら，そのようなアプローチでは C++ の良い点を取り入れることができなかったり，つまずいたときにどこが悪いのかがわからないからです．

また，Visual C++ や C++ Builder を使いこなして充実した仮想システムを作るには，この学習パターンだけでは足りません．書店にあるたくさんの参考書籍の中から自分の能力にあったものを探してきて読むことが必要です．

るサービスをあらかじめ用意し考慮するようなシステムにしておけば，どのようなフィルタを設計してもフィルタの派生クラス以外の部分を変更する必要はまったくありません．

システムの中の重要な機能モジュールを分離し，基底クラス，派生クラスの関係を作っておけば，このように他の部分への影響を考えることなく，新しい試みを安全に試すことができます．

● CPUの変更について

仮想システムから実システムへの移行が成功したら，実装するターゲットシステムのCPUが変更になっても，クラス化した部分のコンポーネントは新しいCPUへの移行も可能なはずです．このとき変更が必要なのは，CBehaviorOnRealSystemクラスに相当する部分だけです．このクラスはCBehaviorOnSystemクラスの派生クラスになっているので，CBehaviorOnRealSystemAやCBehaviorOnRealSystemBを新たに作り，実システムに依存する部分以外の機能をさまざまなプラットホーム間で共有することも可能でしょう．

● C言語でオブジェクト指向プログラミングを行うことについて

C++言語が使える8ビット，16ビット，32ビットのCISC CPUは，まだ少ないと思います．しかし，オブジェクト指向プログラミングの特徴である継承を使わないと，今回のデザインパターンを実現することはできません．日立製作所のCPUであるH8Sと日立純正のコンパイラはC++に対応していますが，他のメーカーのCPUでもC++への対応が可能になってほしいと思います[注8]．

● システムの一部だけにオブジェクト指向プログラミングを使用することについて

組み込みシステム全体のソフトウェアを全部C++で書くことが，現時点で最良な選択肢であるとは思いません．それは，過去のCで作成したソフトウェア資産の流用の必要性や，オブジェクト指向プログラミングを行うことのできる組み込み系のソフトウェア技術者の育成が間に合わないことがよくあるからです．また，C言語で実現できるシステムをC++言語に置き換えることで，若干のメモリ資源の増加や，CPUパフォーマンスの増加の影響は避けられないでしょう．メモリ資源やCPUパフォーマンスがすでにギリギリの状態のシステムでは，システム全体をオブジェクト指向設計することが許されない場合もあると考えられます．

また，実機上のデバッグで継承した派生クラスの中の変数をチェックする際に，デバッガには従来のC言語では必要のなかった名前マングリングを使って実際に使われている変数のありかを探し出さなければならないため，デバッガシステムには新しい機能が要求されます．Visual C++やC++ Builderには当然用意されているこの機能が，ICEには用意されていない場合もあります．

おわりに

オブジェクト指向のデザインパターンである「Template Methodパターン」と「Strategyパターン」を使って，信号計測におけるフィルタの検討と，実装前のシミュレーションの実現，およびリアルシステムへのスムースな移行を実現しました．

この手法は，リアルタイム組み込みシステムのきびしい制約条件を十分に理解したうえで，オブジェクト指向設計をどのように利用できるかを検討した一例です．

また，この方法は，組み込みシステムの安全性や信頼性をビジュアルな方法で確認できるだけでなく，一般向けのVisual C++やC++ Builderなどの安価な開発環境および，それらに関する書籍や，フリーのクラスライブラリなどを活用できるために，組み込み系の高価な開発環境に頼ることなく安価に検証が行えるという利点もあります．

この例を通じて，ご自分の組み込みシステムの中の何を「オブジェクト」としてとらえ，どの部分にデザインパターンを利用するのかを考えていただきたいと思います[注9]．

参考文献

1) 渡辺博之，渡辺政彦，堀松和人，渡守武和，『組み込みUML(eUMLによるオブジェクト指向組み込みシステム開発)』，(株)翔泳社
2) 山下浩，黒羽裕章，黒岩健太郎，『C++プログラミングスタイル』，(株)オーム社
3) Erich Gamma，Richard Helm，Ralph Johnson，John Vlaissides共著，本位田真一/吉田和樹 監訳，『オブジェクト指向のおける再利用のためのデザインパターン』，ソフトバンク(株)
4) スティーブン・R・デイビス著，瀬谷啓介訳，『改訂C++のからくり』，ソフトバンク(株)

さかい・よしお/まつざわ・わたる

注8：京都マイクロコンピュータ(株)のC/C++コンパイラ“exeGCC”が組み込み系のいくつかのマイコンに対応している．

注9：本稿で解説したサンプルプログラムは，本誌Webページの「ダウンロードのページ」(http://www.cqpub.co.jp/interface/download/contents.htm)からダウンロードできる．また，次の本誌2003年2月号付属CD-ROM「InterGiga No.29」にも収録する予定．

ハッカーの常識的見聞録 ㉕

今月の常識 ヘッドホンでも5.1chバーチャルサウンドをコードレスで楽しもう　■広畑由紀夫

☆パイオニアよりXboxデザインモデルの「ディジタルコードレスサラウンドヘッドホン」が発売された．先行してソニーからも同様の製品が発売されている．今回はそれらの比較検討と，ソニー製ディジタルコードレスサラウンドヘッドホンを購入し，実体験してみることにした．

● Xboxデザイン，ディジタルコードレスサラウンドヘッドホン「SE-XB1」

パイオニアから発売された「SE-XB1」は，Xboxとのデザインの統一感や，その利便性からとてもよくできた製品だと思います．とくに価格帯も4万円台と，先行して発売されたソニーの「MDR-DS8000」よりも安く，5.1chを楽しむことができるのが魅力です．

仕様上の詳細では明記されていませんが，DTS-ESの再生はおそらく5.1chの互換モードで再生しているのでしょう．実際に6.1chサラウンドシステムで聴くのではなく，バーチャルサラウンドヘッドホンとして仮想的に聴いているわけですから，値段から考えてもXboxにDVD再生キットをつけてDVDを楽しんだり，Xboxでゲームをプレイする際の効果などを得られることは非常にお得だと思います．

S端子とバーチャルサラウンドヘッドホンで深夜に映画を楽しんだり，映画を観たらそのままゲームを楽しむなど，コードレスであることの利点は非常に高いと思います．赤外線を使ったコードレスなので無線ほどではないのですが，映画を見ている最中に飲み物を取りに行くときも移動するためにヘッドホンを外さなくても済む，ソファで寝そべっているときに寝返りを打つなどでコードに引っ張られることがないという点は，深夜ぐらいにしか映画を観たり，ゲームをプレイする時間のない人に，快適な環境を提供してくれることでしょう．

● ディジタルサラウンドヘッドホンシステム『MDR-DS8000』

2001年11月にソニーから発売された「ディジタルサラウンドヘッドホンシステム」です．この製品のいちばんの売りは6.1chバーチャルサラウンドです．センターサラウンドスピーカのシミュレーションで，音の定位の向上がとてもよく図られています．とくに，アクション映画のDVDをDTSで再生するときに，その効果がよく出ています．

そのほかには，Gyrotrak機能で横を向くと，頭の動きをトレースして前方の音が横から聞こえるようになるという，音の再生方向を外部スピーカで再生しているように聴くことができるオプションが気に入っています．今回は，5.1chと6.1chの音の定位効果を調べるのにとくに役立ちました．

● DTSの5.1ch/6.1chバーチャルサラウンド効果

今回，DTSフォーマットで提供されている「BEHIND ENEMY LINE」のDTSトラックを，DOLBY DIGITAL，DTS 5.1，DTS 6.1の各バーチャルサラウンドで実際に聴いてみました．

〔写真1〕MDR-DS8000

部屋にスピーカを置いているのではなく，あくまでもヘッドホンで仮想的に再現されているわけなので，6.1chと5.1chの違いは，はっきり識別しにくいものの，DTSトラックの映画を再生したときの音の方向性については，個人差はあるにしても効果は出ていると思いました．Gyrotrackモードで，5.1chと6.1chを切り替えながら音の定位を360度ぐるりと回りながら聴いてみると，そうした細かな点が実際に感じやすいと思いました．5.1chでは，後方からの音の定位がややぼやけたように感じたので，6.1chの効果は確かにあると思います．

ただ，6.1chのバーチャル効果を本格的に体験できるようになるには，DTS-ESディスクリートでソフトが提供されるまで，効果がいまーつ薄い感じです．

● 購入の決め手

筆者は，おもにDVDの再生をメインにおいて購入を考えたので，6.1chバーチャルサラウンドのMDR-DS8000を購入しましたが，XboxでのゲームプレイとDVD再生を考えるのであれば，SE-XB1はとてもお得だと思います．金額的に見てもMDR-DS8000を購入するなら，XboxとSE-XB1がセットで買えてしまうくらいの価格(9月末時点での比較)ということも重要だと思います．

実際に，DTS5.1chモードで，XboxでのDVD再生を行ってみましたが，さすがに専用DVDプレーヤがいらないくらいに鮮明で，DTSサウンド効果もヘッドホンとしては素晴らしいものでした．SE-XB1はXboxでのプレイ用にチューニングされているという発表からも，Xboxと組み合わせての使用に適しているものと思われます．

近年のサウンドの高品質化にしたがって，これからももっと高品位かつ多用途のサラウンドヘッドホンなどが発売されてくることでしょうし，また組み込みでのサウンド関連での光ディジタル端子搭載がもっと利用されはじめる日も近いのかもしれません．

ひろはた・ゆきお　OpenLab.

コンフィギュラブルプロセッサ「Xtensa」を使ったFIRフィルタの高速化

永峰 譲

命令の種類やバス幅，動作クロックなどの要素をニーズにあわせてカスタマイズ可能なプロセッサコア「Xtensa」〔開発・販売：テンシリカ（株）〕について，前編（本誌2002年12月号）では概要を解説した．後編となる今回は，Xtensaを使ったFIRフィルタの高速化についてくわしく解説する．（編集部）

FIRフィルタの高速化

一般的に16ビットのデータを扱うFIRフィルタのC言語によるプログラムは，**リスト1**のようになります．ここではタップ数を256個として記述しています．x[i]とy[i]を乗算しansに加算する処理になります．

1.1 Mac16オプション

Xtensaの命令オプションにMac16（積和演算器）オプションがあります．このプログラムをMac16オプションを追加したXtensa用にコンパイルすると，**リスト2**のようなアセンブラコードが出力されます．

積和演算をループさせるために，Xtensaのハードウェアループで構成されたゼロオーバヘッドループを使用して7行目のloopnez命令が実行されます．ループ終了後にL1にジャンプするので，ここでは全部で五つの命令が一つのループで実行されます．命令は**図1**のように実行されます．

実際にデータフローを**図2**（次頁）に沿って確認してみます．T1でl16ui命令が実行されます．これはアドレスレジスタのポイントするメモリ領域から16ビットデータをアドレスレジスタにロードする命令です．ここではa2がポイントする領域のデータをa5にロードしています．しかし，Xtensaのロード命令は2サイクル命令であるため，T1ではデータがロードできずT2でロードされます．T2ではもう一方の16ビットデータをa3のポイントする領域からa4にロードしています．次のmula.aa.ll命令がT3で実行されるはずですが，T3ではa4のデータがロードされていません．したがって，Xtensaはこの命令を実行できないためにstallします．プログラムとしてはnopなどを入れる必要はなく，Xtensaのハードウェアがケアします．したがって，T4でmula.aa.ll命令は実行されます．T5でadd.n命令が実行されa3のポインタ値を2（short型）バイト分インクリメントします．T6では同様にa3をインクリメントします．そしてループ先頭に戻って以上の演算を繰り返すことになります．

6サイクルで一つの積和演算を実行できることになります．実はMac16オプションを追加したXtensaの拡張ALUにある乗算器は，1サイクルで積和演算を可能にする構成になっています．しかし，ここではあえてFIRフィルタ演算を高速化するためのTIE命令を設計しながら「どのようにして処理能力の高い命令を設計するか」を説明します．

1.2 TIE命令による積和演算器

最初に，積和演算を行うためのTIE命令を設計します．前節で出力されたコンパイル結果をもとにして命令を作成します．積和演算に必要な命令は，二つの16ビットデータのロード，乗算，アキュムレータへの加算，アドレスポインタのインクリメントとなります．また，データを保持するためのリソースをアドレスレジスタにするか，ステートレジスタにするか，あるいはTIE命令用拡張レジスタファイルにするかを決定しなければなりません．

〔リスト1〕16ビットのデータを扱うFIRフィルタのCプログラム

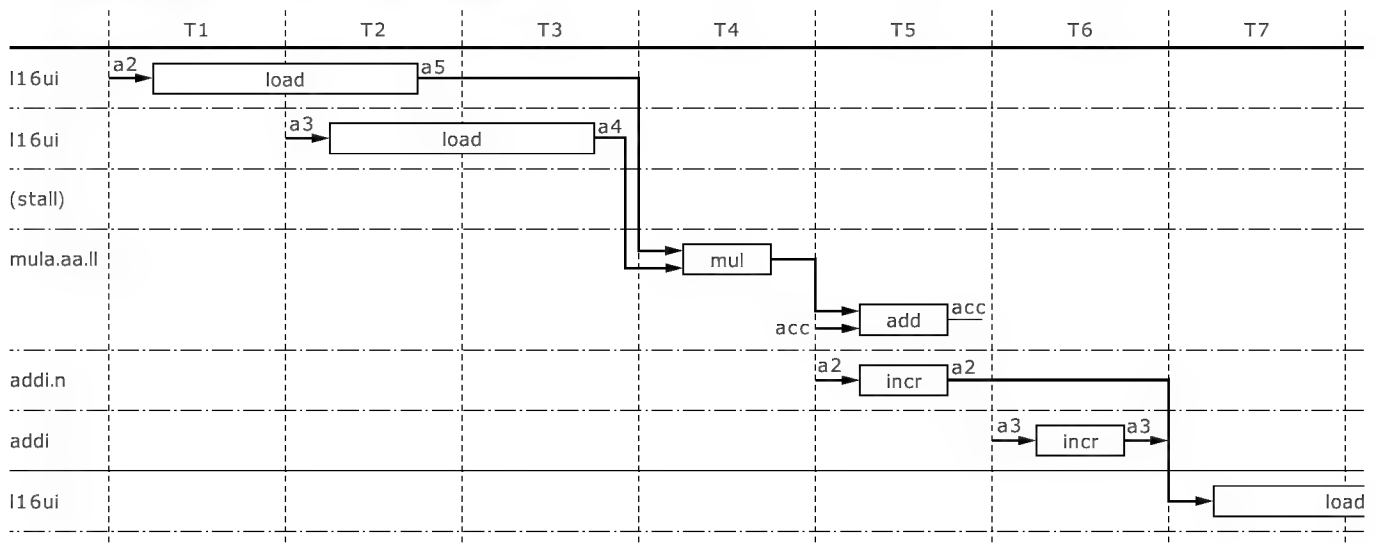
```
int fir org(short *x, short *y)
{
    int i;
    int ans;

    ans = 0;

    for(i=0;i<256;i++)
    ans += x[i] * y[i];

    return(ans);
}
```

〔リスト2〕Mac16（積和演算器）オプション

```
 1  <fir org>
 2              entry a1, 32
 3              movi.n a6, 0
 4              wsr a6, 16
 5              movi a4, 256
 6              nop.n
 7              loopnez a4, L1
 8              l16ui a5, a2, 0
 9              l16ui a4, a3, 0
10              mula.aa.ll a5, a4
11              addi.n a3, a3, 2
12              addi a2, a2, 2
13  L1:
14              rsr a2, 16
15              retw
```

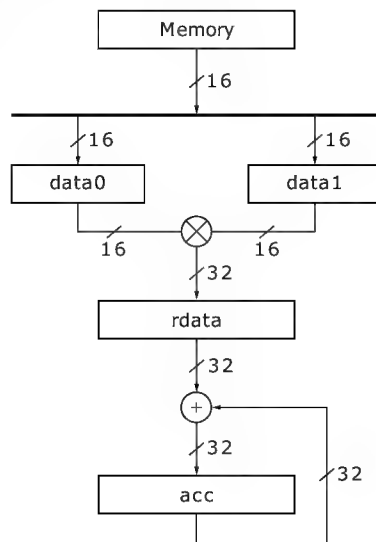
〔図1〕コンパイラによるコードの実行フロー

Cycle Count		
0	l16ui	1st loop
1	l16ui	
2	(stall)	
3	mula.aa.ll	
4	add.n	
5	add	
6	l16ui	2nd loop
7	l16ui	
8	(stall)	
9	mula.aa.ll	
10	add.n	
11	add	
12		

〔図2〕Mac16命令によるパイプラインデータフロー動作

ここでは拡張性を考慮してステートレジスタを使用して設計します．**図3**のような構成を考えます．命令はステートレジスタdata0とdata1にデータをロードする命令，ステートレジスタdata0とdata1を乗算し結果をrdataに保持する命令とアキュムレータacc32にrdataの値を加算する命令が必要になります．必要なステートレジスタの定義をしなければなりません．

```
state data0 16
state data1 16
state rdata 32
state acc32 32
```

次にこれらのステートレジスタをデバッグ時に可視化できるようにユーザーレジスタに次のようにして割り当てます．

〔図3〕TIE命令による積和演算モジュールブロック図

```
user_register DATA0 0 {data0[15:0]}
user_register DATA1 1 {data1[15:0]}
user_register RDATA 2 {rdata[31:0]}
user_register ACC32 3 {acc32[31:0]}
```

メモリからのロード命令を設計するためにインターフェース信号の定義が必要です．これはロードするデータ幅に合う信号名を定義します．16ビットデータを扱うため，16ビット幅のインターフェース信号とアドレス用のインターフェース信号が必要になります．次のように記述します．

```
interface VAddr 32 core out
interface MemDataIn16 16 core in
```

設計する命令に必要なリソースの定義ができました．次に，実際に必要な命令をロード命令から設計します．

```
opcode ldata0 op2=4'b0000 LSCX
opcode ldata1 op2=4'b0001 LSCX
```

命令名はldata0とldata1とします．ここで，メモリアクセス用に使用できるsub-opcodeクラスはLSCX（あるいはLSCI）です．iclass[注1]の定義は，ステートレジスタの入出力を3番目の{}に定義します．また，メモリとのインターフェース信号を次の4番目の{}に定義します．次のように記述します．

```
iclass ld0 {ldata0} {in ars} {out data0}
                        {out VAddr, in MemDataIn16}
iclass ld1 {ldata1} {in ars} {out data1}
                        {out VAddr, in MemDataIn16}
```

命令機能内容は，アドレスレジスタでポイントしているメモリ領域から16ビットのデータをロードするので，**リスト3**のように定義できます．

注1：iclass sectionで最初に記述する{ }のない名前はグループ名．命令の入出力が同じ命令は，グループ化して記述できる．たとえば，Xtensaの実際の命令ではadd命令，sub命令，and命令やor命令のように，アドレスレジスタを二つ入力して一つ出力する命令群のようなものを一つのグループにできる．

〔リスト3〕命令機能内容

```
reference ldata0 {
        assign VAddr = ars;
        assign data0 = MemDataIn16;
}

reference ldata1 {
        assign VAddr = ars;
        assign data1 = MemDataIn16;
}
```

〔リスト4〕ロードのタイミング定義

```
schedule ld0 {ldata0} {
        def data0 2;
}

schedule ld1 {ldata1} {
        def data1 2;
}
```

〔リスト5〕乗算結果をrdataに保持する

```
opcode multie op2=4'b0000 CUST0

iclass mlt {multie} { } {out rdata, in data0, in data1}

reference multie {
        wire s = 1'b1;
        wire [31:0] sum0 = TIEmul(data0, data1, s);
        assign rdata = sum0;
}
```

〔リスト6〕アキュムレータacc32にrdataを加算する命令

```
opcode addacc op2=4'b0001 CUST0

iclass ada {addacc} { } {inout acc32, in rdata}

reference addacc {
        assign acc32 = acc32 + rdata;
}
```

ところが，Xtensaのロード命令は2サイクル命令[注2]であるため，ステートレジスタdata0とdata1のロードのタイミングを定義しなければなりません．ここでは2サイクルであるため**リスト4**のように定義できます．

次に，data0とdata1の乗算結果をrdataに保持する命令を設計します(**リスト5**)．ここで，iclassの定義でオペランドを使用しないでステートレジスタのみ定義する場合には，オペランド用の{ }は省略できません．またreferenceではTIEmulという，Xtensaのビルトインモジュール[注3]を使用します．次は，アキュムレータacc32にrdataを加算する命令を設計します(**リスト6**)．以上で命令設計ができました．続いて，命令をCソース上に組み込みコンパイルします(**リスト7**)．**リスト8**のようなアセンブラコードに展開されます．

実際に，データフロー(**図4**)を確認し何サイクルで積和演算

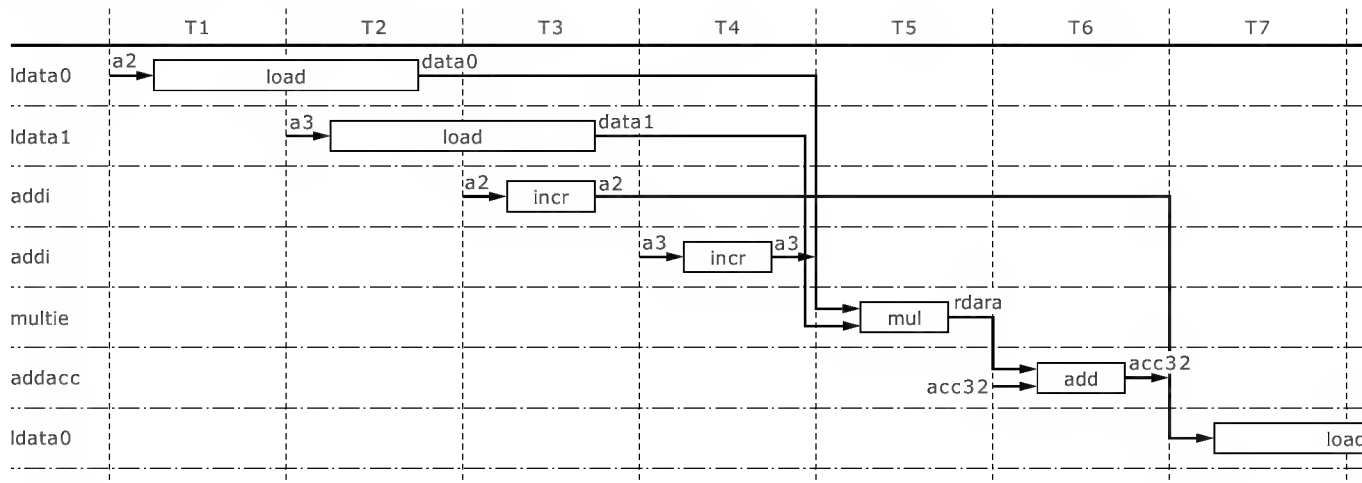
〔リスト7〕命令をCソース上に組み込みコンパイル

```
int fir tie1(x, y)
        short *x;
        short *y;
{
        int i;

        WPDATA(0);
        WACC32(0);

        /* Calculation of FIR
                        filter */
        for(i=0;i<TAP;i++)
        {
                ldata0(x);
                ldata1(y);
                multie();
                addacc();
                x++;
                y++;
        }

        return(RACC32());
}
```

〔リスト8〕展開されたアセンブラコード

```
1   <fir tie1>:
2           entry a1, 32
3           movi.n a4, 0
4           wpdata a4
5           wacc32 a4
6           movi a4, 256
7           loopnez a4, L1
8           ldata0 a2
9           ldata1 a3
10          addi.n a2, a2, 2
11          addi a3, a3, 2
12          multie
13          addacc
14  L1:
15          racc32 a2
16          retw
```

が実行されているか確認します．ループは**リスト8**の8行目のldata0命令から13行目のaddacc命令までの6命令です．T1でldata0命令が実行されてa2でポイントされたデータをdata0に，そしてT2でldata1命令が実行されてa3でポイン

〔図4〕パイプラインデータフロー動作(1)

注2：Xtensaのロード命令は2サイクル命令として実行されている．最初の実行サイクルステージでアドレス演算を行い，次のサイクルでデータのロードが完了する．

注3：TIE命令を設計する際に，Xtensaのオプショナル命令と同等のハードウェアで構成されたビルトインモジュールがある．現在用意しているビルトインモジュールは乗算器，積和(差)演算器，キャリイン付加算器とキャリセーブ加算器．

トされたデータをdata1にロードします．T3ではaddiが実行されポインタa2のアップデートが実行されます．addi命令はldata1で出力されるdata1を使用しないため，パイプラインはstallしません．T4でa3のアップデートが実行されます．T5ではdata0とdata1はすでにロードされているため，multie命令が実行されます．T6ではアキュムレータへの加算命令が実行されます．以降，同様のデータフローで積和演算が実行され，**6サイクルで一つの積和演算**が実行されることになります．

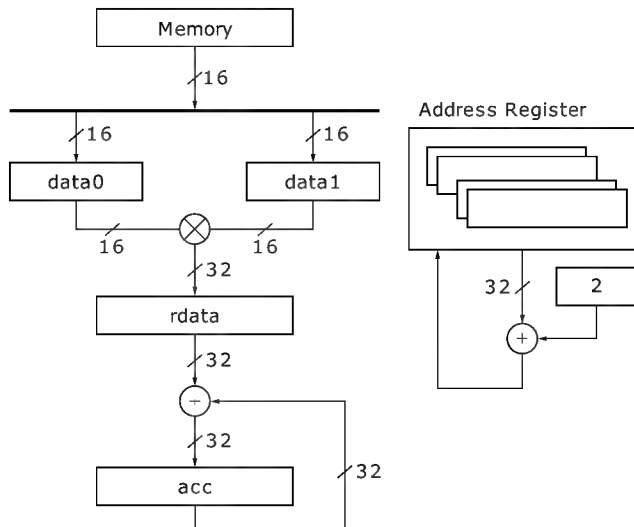
1.3 アドレスアップデート付き積和演算器

前節ではアドレスのアップデートをコンパイラが出力したコード処理させる命令を作成しました．ところが，データのロードを行いながらポインタのアップデートを実行できれば，1積和演算に必要なサイクル数が軽減できます．そこで，ロード命令にアドレスのアップデート機能を追加します．アドレスのアップデートは，一つのデータが16ビットのshort型であることから2バイト分インクリメントすればよいことになります．**図5**のような構成を考えます．前節の命令との違いは，ldata0とldata1のポインタ用アドレスの定義のみとなります．動作としては，アドレス用のポインタを2バイト分インクリメントして返せばよいことになります．したがって，iclassのアドレスレジスタの入出力が次のように変更されます．

```
iclass ld0 {ldata0} {inout ars} {out data0}
                    {out VAddr, in MemDataIn16}
iclass ld1 {ldata1} {inout ars} {out data1}
                    {out VAddr, in MemDataIn16}
```

arsがinoutと定義されることで，命令実行後に更新されることになります．次に命令機能の変更があります．ldata0とldata1命令で2バイト分アドレスを更新しなければなりません．**リスト9**のように定義できます．

一方，アドレスレジスタの更新は1サイクルで実行できれば効率が良いため，スケジュールの変更も行います(**リスト10**)．

〔**図5**〕**アドレスアップデート付積和演算モジュールブロック図**

以上で変更が終了しました．続いてTIE命令をCソース上に組み込み，コンパイルします．このとき，すでにアドレスのアップデートはロード命令に組込まれているためポインタの更新は，**リスト11**のように必要ありません．**リスト12**のようなアセンブラコードに展開されます．

実際に，**図6**のデータフローを確認し，何サイクルで積和演算が実行されているかを確認します．ループは**リスト12**の8行目のldata0命令から11行目のaddacc命令までの4命令です．T1でldata0命令が実行されてa2でポイントされたデータをdata0にロードし，a2のポインタ値をアップデートします．次にT2でldata1命令が実行されてa3でポイントされたデータをdata1にロードし，a3のポインタ値をアップデートします．T3でmultie命令を実行したいけれどdata1がロードされていません．したがってパイプラインがstallします．T4でmultie命令が実行され，T5でアキュムレータへの加算が行われます．つまり，**5サイクルで一つの積和演算**が実行されることになります．

高速化を考える場合には，命令をSIMD化する手法と，複数の命令機能を1命令に統合してデータフローをパイプライン化する手法があります．最初に命令のSIMD化での高速化を行い，次に機能パイプライン化命令を設計する方法です．

1.4 SIMD命令による高速化

命令をSIMD化し演算器を増すことでフィルタの高速化を行います．この手法はさまざまなアプリケーションに応用できます．とくに，データのロードストアのバンド幅を上げることでデ

〔**リスト9**〕**命令機能の変更**

```
reference ldata0 {
        assign VAddr = ars;
        assign data0 = MemDataIn16;
        assign ars = ars + 2;
}

reference ldata1 {
        assign VAddr = ars;
        assign data1 = MemDataIn16;
        assign ars = ars + 2;
}
```

〔**リスト10**〕**スケジュールの変更**

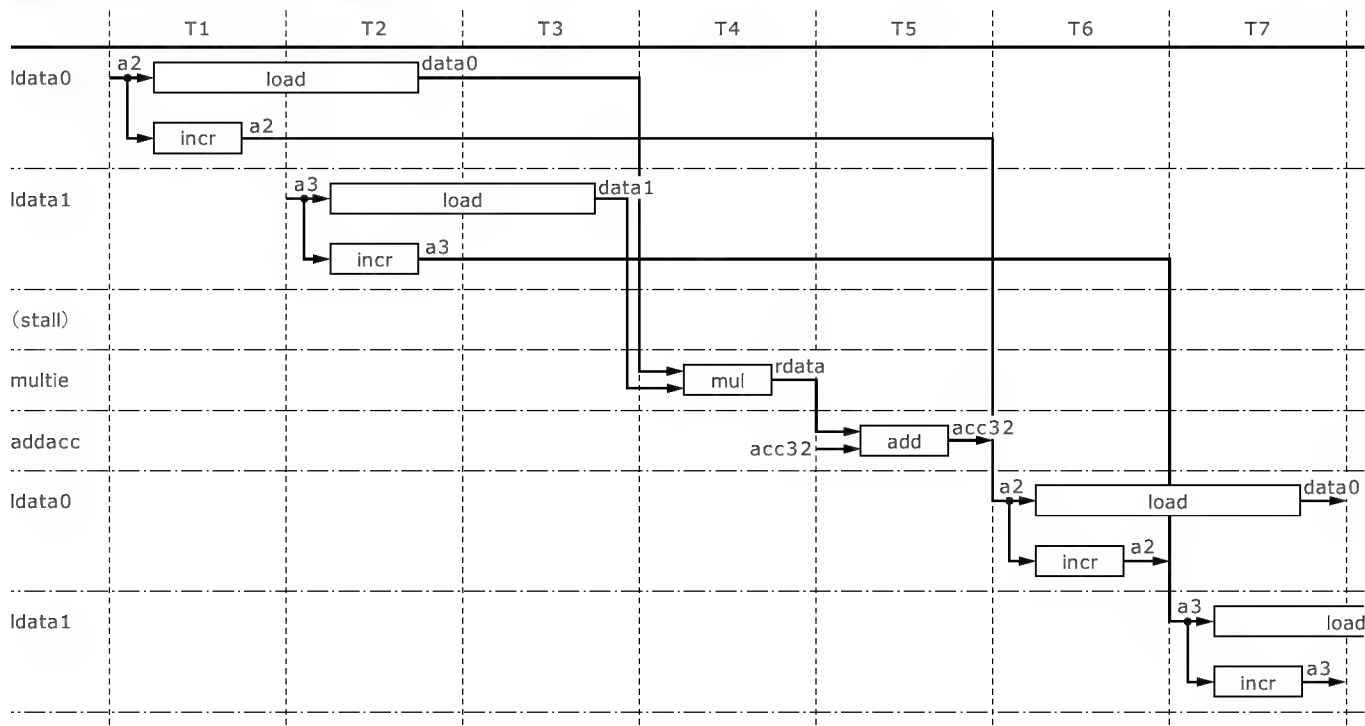
```
schedule ld0 {ldata0} {
        def ars 1;
        def data0 2;
}

schedule ld1 {ldata1} {
        def ars 1;
        def data1 2;
}
```

〔**リスト11**〕**TIE命令をCソース上に組み込んでコンパイル**

```
int fir tie2(x, y)
        short *x;
        short *y;
{
        int i;

        WPDATA(0);
        WACC32(0);

/* Calculation of FIR filter */
        for(i=0;i<TAP;i++)
        {
                ldata0(x);
                ldata1(y);
                multie();
                addacc();
        }

        return(RACC32());
}
```

〔**リスト12**〕**展開されたアセンブラコード**

```
 1  <fir tie2>:
 2              entry a1, 32
 3              movi.n a4, 0
 4              wpdata a4
 5              wacc32 a4
 6              movi a4, 256
 7              loopnez a4, L1
 8              ldata0 a2
 9              ldata1 a3
10              multie
11              addacc
12  L1:
13              racc32 a2
14              retw.n
```

〔図6〕パイプラインデータフロー動作(2)

ータのロード/ストアの回数を減らし，データのスループットを上げることができます．Xtensa の TIE 命令の強力な機能の一部といえます．

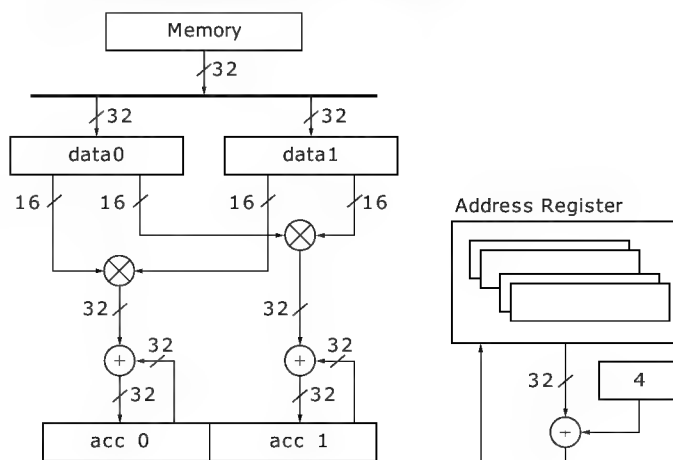
● 2並列SIMDの積和演算器

FIR フィルタのように連続したデータを積和演算するようなアプリケーションでは，SIMD 命令を容易に設計できます．積和演算で扱うデータは16ビットのshort型です．二つの連続したデータでは合計32ビットとなります．つまり32ビットデータを取り込めば，二つのshort型データを取り込むことができます．これを一つの命令で二つの積和演算が実行できるようにすることで，2並列SIMD演算が実行できます．**図7**のような二つの積和演算を実行できる命令を作成します．ロード命令にはアドレスのアップデート機能を追加します．アップデートの値は二つのshort型分の4バイトのインクリメントになります．ステートレジスタでの定義はdata0とdata1は32ビットとなり，アキュムレータはacc_0とacc_1の二つを用意します．

```
state data0 32
state data1 32
state acc_0 32
state acc_1 32
```

それぞれをユーザーレジスタに定義します．

```
user_register 0 data0
user_register 1 data1
user_register 2 acc_0
user_register 3 acc_1
```

32ビットのデータをメモリからロードするため，インターフェース信号を32ビットで定義します．

```
interface VAddr        32 core out
interface MemDataIn32 32 core in
```

最初にロード命令を設計します．

```
opcode ldata0 op2=4'b0000 LSCX
opcode ldata1 op2=4'b0001 LSCX
```

iclassではインターフェース信号は32ビットを使用します．

```
iclass ld0 {ldata0} {inout ars} {out data0}
                    {out VAddr, in MemDataIn32}
iclass ld1 {ldata1} {inout ars} {out data1}
                    {out VAddr, in MemDataIn32}
```

〔図7〕2 SIMDの積和演算モジュールブロック図

〔リスト13〕機能定義

```
reference ldata0 {
        assign VAddr = ars;
        assign data0 =
                      MemDataIn32;
        assign ars = ars + 4;
}

reference ldata1 {
        assign VAddr = ars;
        assign data1 =
                      MemDataIn32;
        assign ars = ars + 4;
}
```

〔リスト14〕2サイクル命令

```
schedule ld0 {ldata0} {
        def data0 2;
}

schedule ld1 {ldata1} {
        def data1 2;
}
```

〔リスト15〕命令の機能

```
reference macx02 {
        wire s = 1'b1;
        wire n = 1'b0;
        wire [31:0] sum0 = TIEmac(data0[15:0], data1[15:0],
                                          acc_0[31:0], s, n);
        wire [31:0] sum1 = TIEmac(data0[31:16], data1[31:16],
                                          acc_1[31:0], s, n);
        assign acc_0 = sum0;
        assign acc_1 = sum1;
}
```

〔リスト16〕2並列のSIMDの積和演算命令

```
int fir_tie11(x, y)
        short *x;
        short *y;
{
        int i;
        int ans;

        WUR0(0);
        WUR1(0);

        /* Calculation of FIR
                    filter */
        for(i=0;i<128;i++)
        {
                ldata0(x);
                ldata1(y);
                macx02();
        }

        ans = RUR(2) + RUR
                          (3);

        return(ans);
}
```

〔リスト17〕アセンブラコード

```
 1  <fir tie11>
 2          entry a1, 32
 3          movi.n a4, 0
 4          wur0 a4
 5          wur1 a4
 6          movi a4, 128
 7          loopnez a4, L1
 8          ldata0 a2
 9          ldata1 a3
10          macx02
11  L1:
12          rur2 a4
13          rur3 a2
14          add.n a2, a4, a2
15          retw.n
```

機能定義では二つのshort型データを演算に使用するためアドレスのインクリメント値が4になります(**リスト13**).

メモリからのロード命令であるから2サイクル命令となります(**リスト14**).

次に積和演算の命令です.

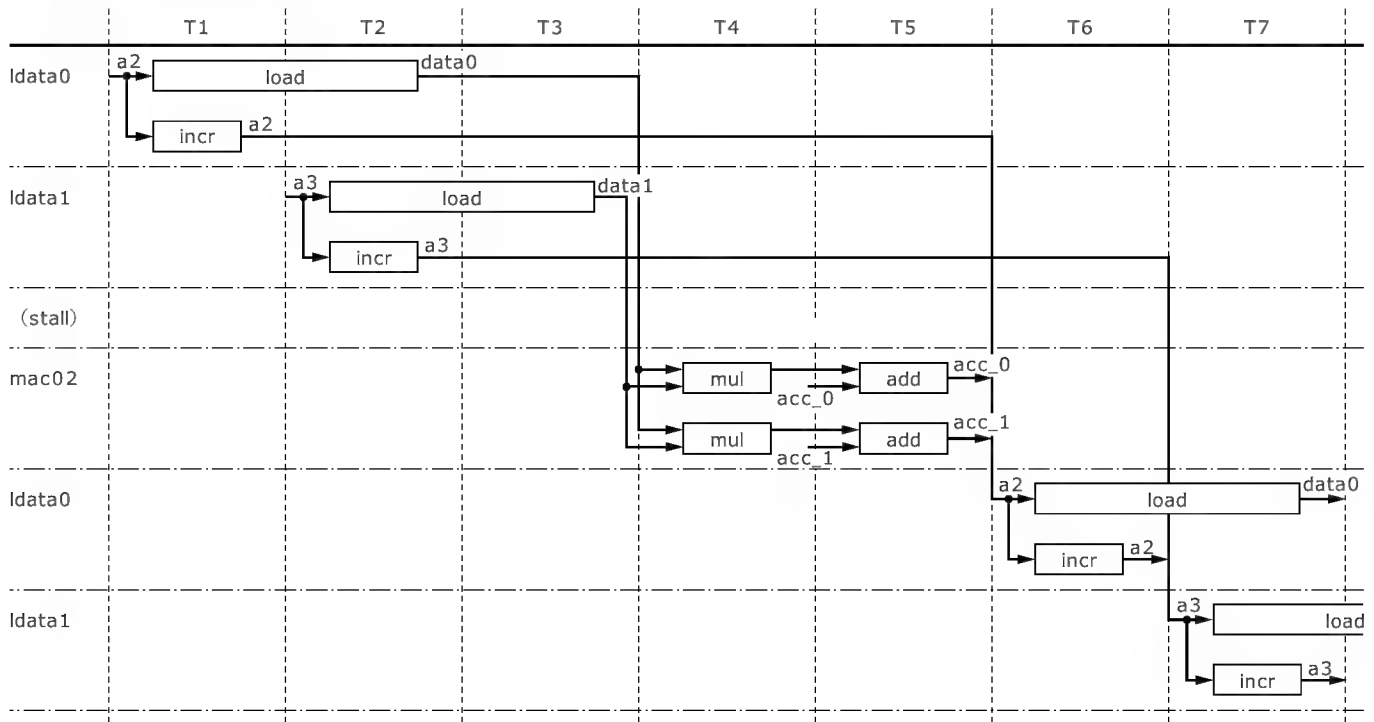
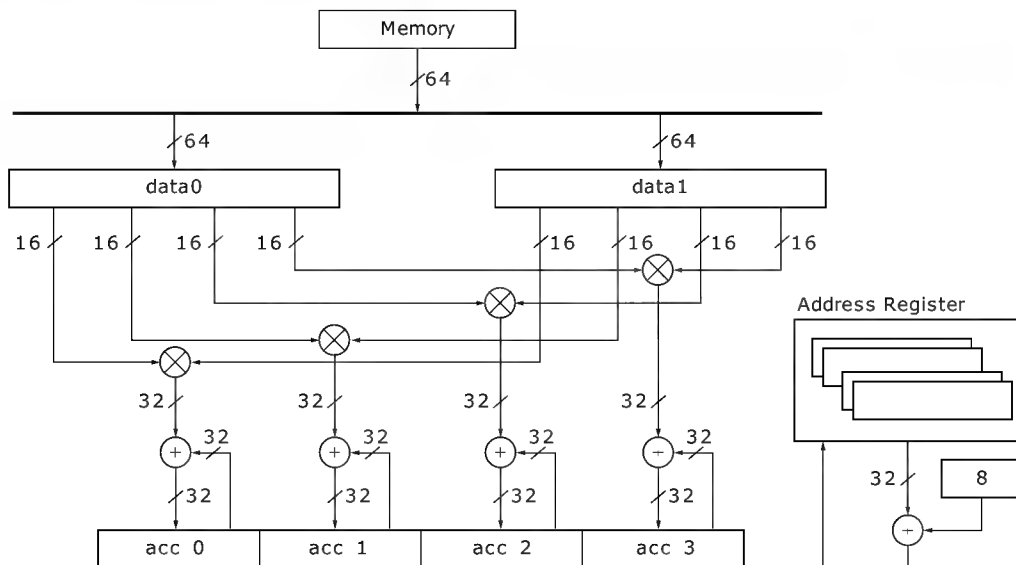
```
opcode macx02 op2=4'b0000 CUST0
```

ここでは乗算と加算を一つの命令で設計します. iclassではアキュムレータが増えるので, 次のようになります.

```
iclass mc2 {macx02} { } {inout acc_0,
              inout acc_1, in data0, in data1}
```

命令の機能は**リスト15**のようになり, ここではXtensaのビルトインモジュールTIEmacを使用して設計します. 2並列なので, 積和演算器が二つ搭載されます.

macx02命令は, 2サイクル命令(命令実行機能を2分割するにはRTL合成時にリタイミング機能を必要とする)で設計します. 2サイクル命令にすることで2サイクル目を実行している間に次のロード命令を実行することができます. またアキュムレータに加算するタイミングは2サイクル目で問題ないために, 次のように, acc_0とacc_1の入力は2サイクル目で, 出力も2

〔図8〕パイプラインデータフロー動作(3)

〔図9〕
4SIMDの積和演算モジュール
ブロック図

サイクル目で定義することができます．

```
schedule mc2 {macx02} {
use acc_0 2;
use acc_1 2;
def acc_0 2;
def acc_1 2;
}
```

以上で2並列のSIMDの積和演算命令が設計できました．Cソースコードは**リスト16**のようになります．

2並列SIMDのため，ループ回数は半分の128回となります．また，ループ計算が終了した後acc_0とacc_1の内容を加算しなければなりません．**リスト17**のようなアセンブラコードに展開されます．

実際に，**図8**のデータフローをもとにし，何サイクルで積和演算が実行されているか確認します．ループは**リスト17**の8行目のldata0命令から10行目のmacx02命令までです．T1でa2のポイントするデータをdata0にロードし，a2のアドレスをアップデートします．続いてT2でa3のポイントするデータをdata1にロードしa3のアドレスをアップデートします．data1がT3でロードされるため，macx02命令はT4で実行されます．T5でmacx02命令の加算部分が実行されますが，同時にldata0命令が実行されます．したがって4サイクルで二つの積和演算が実行されます．つまり**2サイクルで一つの積和演算**が実行されることになります．

● 4並列SIMDの積和演算器

ここでは，積和演算を4並列SIMDにした場合どのようになるか考えてみます．4並列SIMDということで，short型が四つで64ビットの連続したデータが必要になります．Xtensaの場合，PIF（Peripheral Interface）を64ビット以上の構成にすることで64ビットのメモリアクセスが可能になります．この特徴を生かして4並列SIMDの命令を設計します（**リスト18**）．**図9**のような四つの積和演算を実行できる命令を作成します．ステートレジスタの定義は，data0とdata1は64ビットで定義されます．またアキュムレータは4並列のため，四つ必要です．

```
state data0 64
state data1 64
state acc_0 32
state acc_1 32
state acc_2 32
state acc_3 32
```

ここでユーザーレジスタに定義しますが，ユーザーレジスタに定義できるビット数は32ビット[注4]までです．したがって，**リスト19**のようにdata0とdata1は分割して定義します．もちろん，すべてのステートレジスタをゼロクリアできる命令を設計することもできますが，ここではユーザーレジスタで定義することでアクセス可能になる専用命令を使用します．

次に64ビットのデータをメモリからロードする命令を設計す

〔リスト18〕命令機能

```
reference macx04 {
        wire s = 1'b1;
        wire n = 1'b0;
        wire [31:0] sum0 = TIEmac(data0[15:0], data1[15:0],
                                              acc 0[31:0], s, n);
        wire [31:0] sum1 = TIEmac(data0[31:16], data1[31:16],
                                              acc 1[31:0], s, n);
        wire [31:0] sum2 = TIEmac(data0[47:32], data1[47:32],
                                              acc 2[31:0], s, n);
        wire [31:0] sum3 = TIEmac(data0[63:48], data1[63:48],
                                              acc 3[31:0], s, n);
        assign acc 0 = sum0;
        assign acc 1 = sum1;
        assign acc 2 = sum2;
        assign acc 3 = sum3;
}
```

注4：ステートレジスタにユーザーレジスタを定義することで，直接ステートレジスタにロード/ストアできる命令が作成される．この命令はアドレスレジスタと直接アクセスする命令であるため，32ビットという制限が発生する．

〔リスト 19〕data0 と data1 は分割して定義

```
user register 0 data0[31:0]
user_register 1 data0[63:32]
user_register 2 data1[31:0]
user_register 3 data1[63:32]
user_register 4 acc_0
user register 5 acc 1
user_register 6 acc_2
user register 7 acc 3
```

〔リスト 20〕機能定義

```
reference ldata0 {
        assign VAddr = ars;
        assign data0 = MemDataIn64;
        assign ars = ars + 8;
}

reference ldata1 {
        assign VAddr = ars;
        assign data1 = MemDataIn64;
        assign ars = ars + 8;
}
```

〔リスト 21〕2 サイクル命令

```
schedule ld0 {ldata0} {
        def data0 2;
}

schedule ld1 {ldata1} {
        def data1 2;
}
```

〔リスト 22〕スケジュール

```
schedule mc4 {macx04} {
        use acc 0 2;
        use acc 1 2;
        use acc 2 2;
        use acc 3 2;
        def acc 0 2;
        def acc 1 2;
        def acc 2 2;
        def acc 3 2;
}
```

〔リスト 23〕4 並列の SIMD 命令

```
int fir tie11(x, y)
        short *x;
        short *y;
{
        int i;
        int ans;

        WUR0(0);
        WUR1(0);
        WUR2(0);
        WUR3(0);

        /* Calculation of FIR filter */
        for(i=0;i<64;i++)
        {
                ldata0(x);
                ldata1(y);
                macx04();
        }

        ans = RUR(4) + RUR(5)
             + RUR(6) + RUR(7);

        return(ans);
}
```

〔リスト 24〕アセンブラコード

```
 1  <fir tie12>
 2          entry a1, 32
 3          movi a4, 0
 4          wur0 a4
 5          wur1 a4
 6          wur2 a4
 7          wur3 a4
 8          movi a4, 64
 9          loopnez a4, L1
10          ldata0 a2
11          ldata1 a3
12          macx04
13  L1:
14          rur4 a2
15          rur5 a4
16          add.n a2, a2,
                 a4
17          rur6 a4
18          add.n a2, a2,
                 a4
19  rur7 a4
20  add.n a2, a2, a4
21  retw.n
```

るので，インターフェース信号も 64 ビットにしなければなりません．ここで注意しなければならないことは，Xtensa の構成で PIF 幅を 64 ビット以上にすることです．PIF 幅が 32 ビットでは 1 サイクルで 64 ビットのデータアクセスはできません[注5]．

```
interface VAddr 32 core out
interface MemDataIn64 64 core in
```

それではロード命令を設計します．

```
opcode ldata0 op2=4'b0000 LSCX
opcode ldata1 op2=4'b0001 LSCX
```

iclass はインターフェース信号が 64 ビットのものを使用します．

```
iclass ld0 {ldata0} {inout ars} {out data0}
                        {out VAddr, in MemDataIn64}
iclass ld1 {ldata1} {inout ars} {out data1}
                        {out VAddr, in MemDataIn64}
```

機能定義ではアドレスのインクリメント値が 8 に変更されます（**リスト 20**）．メモリからのロード命令なので 2 サイクル命令となります（**リスト 21**）．次に積和演算の命令です．

```
opcode macx04 op2=4'b0000 CUST0
```

iclass はさらにアキュムレータが四つに増えるので以下のようになります．

```
iclass mc4 {macx04} { }
{inout acc_0, inout acc_1, inout acc_2,
            inout acc_3, in data0, in data1}
```

命令機能は**リスト 21** のようになります．4 並列なので，積和演算モジュールが四つになります．この命令も 2 サイクル命令にするため，**リスト 22** のようにスケジュールされます．

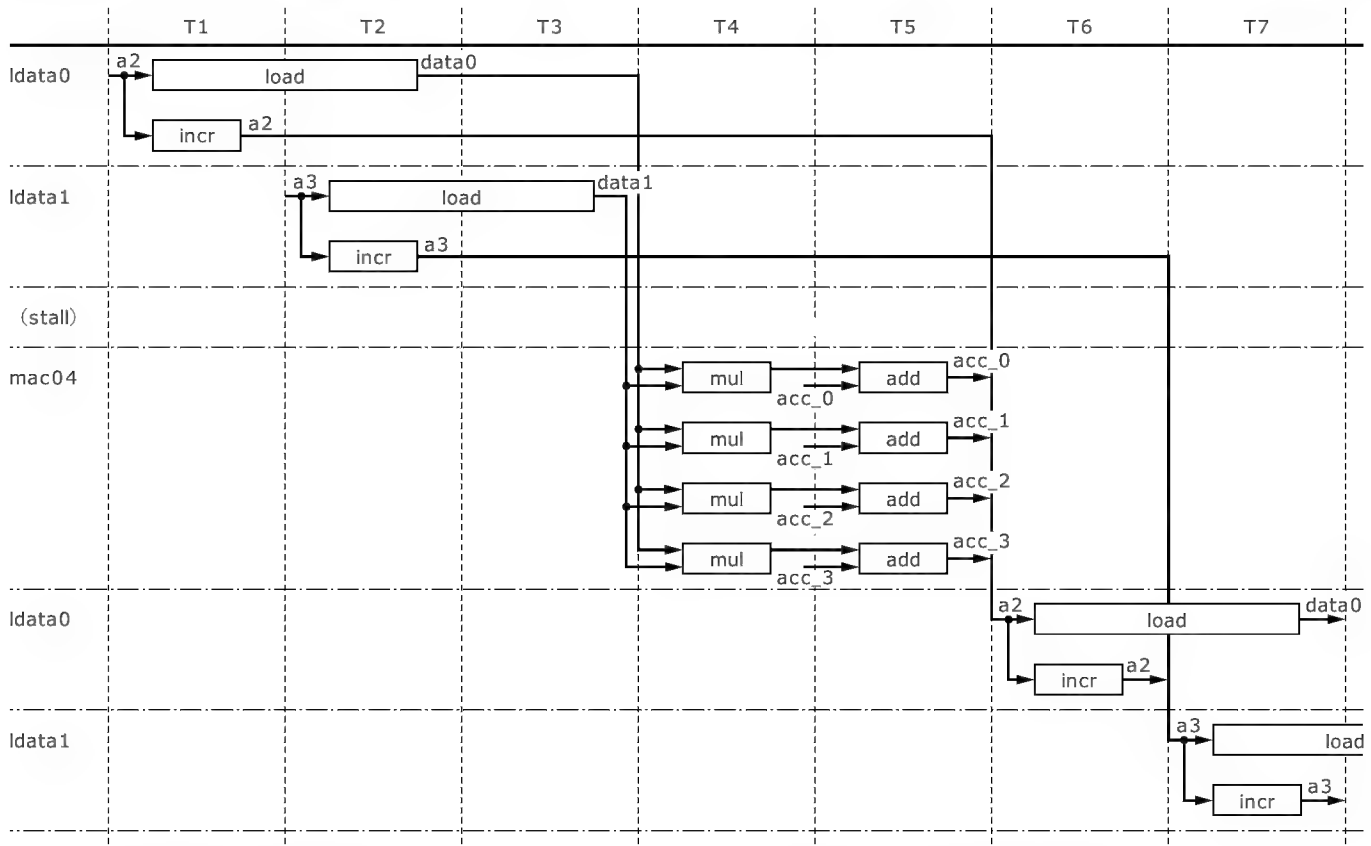
4 並列の SIMD 命令が設計できました．C ソースコードは**リスト 23** のようになります．4 並列 SIMD のため，ループ回数はさらに半分の 64 回となります．また，ループ計算が終了した後 acc_0，acc_1，acc_2 と acc_3 の内容を加算しなければなりません．**リスト 24** のようなアセンブラコードに展開されます．

実際に，**図 10** のデータフローをもとにして何サイクルで積和演算が実行されているか確認します．ループは**リスト 24** の 10 行目の ldata0 命令からから 12 行目の macx04 命令までです．T1 で a2 のポイントする 64 ビットデータを data0 にロードし，a2 のアドレスをアップデートします．アップデートの値は四つの short 型分の 8 バイトです．続いて T2 で a3 のポイントする 64 ビットデータを data1 にロードし，a3 のアドレスを同様にアップデートします．data1 が T3 でロードされるため，macx04 命令は T4 で実行されます．T5 で macx04 命令の加算部分が実行されますが，同時に ldatao 命令が実行されます．したがって 4 サイクルで四つの積和演算が実行されます．つまり，**1 サイクルで一つの積和演算**が実行されることになります．

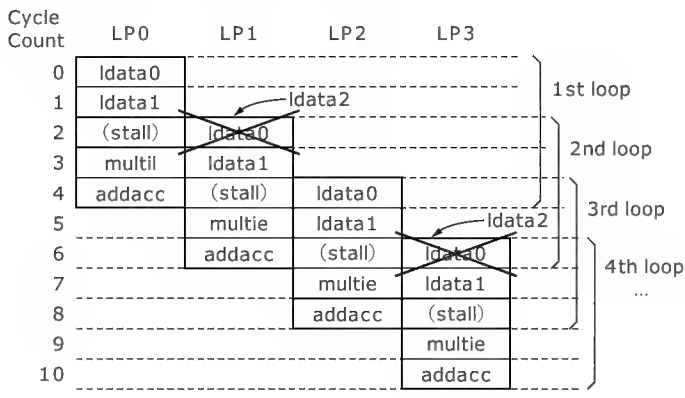
1.5 機能パイプライン化による高速化

一方，積和演算器を増やすことなく性能を上げることはできないのでしょうか？ この問題に対しては，一つの命令で複数の処理をさせて高速化する方法があります．この方法はデータ処理のフローがパイプライン的な動作を行います．実際にどの

注 5：PIF 幅が 32 ビットの場合には，2 回のロードを実行することで必要な 64 ビットデータをロードでき，4 並列の演算を実行できる．このとき演算の速度は 4 倍になるが，ロードの速度は 2 倍となる．

〔図10〕パイプラインデータフロー動作(4)

〔図11〕3機能のパイプライン化の実行フロー

〔リスト25〕data0, data1とdata2の定義

```
state data0 16
state data1 16
state data2 16
state rdata 32
state acc32 32

user_register DATA0 0 {data0[15:0]}
user_register DATA1 1 {data1[15:0]}
user_register DATA2 2 {data2[15:0]}
user_register RDATA 3 {rdata[31:0]}
user_register ACC32 4 {acc32[31:0]}
```

程度の効果があるか確認してみましょう.

● 3機能のパイプライン化

データのロードとアドレスのアップデートのほかに加算あるいは乗算を1命令で実行できる命令を考えてみます. ここでデータのロードには2サイクル必要であることから, 動作順序を確実に追いかける必要があります. ループ中はサイクルごとにロード命令が実行されることが理想となります. そこで**図11**のように必ずメモリからのロード命令が実行されるように実行フローを考えてみます. 一方, ldata0命令のみでは, 乗算命令実行時に必要なdata0を入力できません. LP0で必要なdata0はLP1のldata0でロードされてしまい, 上書きされてしまいます. そこで新しいステートレジスタdata2を定義します. このことで必要なデータを上書きすることは回避できます. 命令を設計するにあたって, どの機能が一つの命令で実行されればよいか考えてみます. ldata0命令はaddacc命令と同一サイクルで実行され, ldata1命令はmultie命令と同一サイクルで実行され, ldata2命令はaddacc命令と同一サイクルで実行できれば, この実行フローどおりにデータが処理されます. ただし, ldata1命令とmultie命令の設計時には乗算ユニットへどのデータを入力するか考慮しなければなりません. これを実際に設計します. 乗算用のデータ用のステートレジスタはdata0, data1とdata2の三つを**リスト25**のように定義します.

インターフェース信号は, 16ビットのロード命令を設計することから次のようになります.

```
interface VAddr 32 core out
interface MemDataIn16 16 core in
```

ldata0命令とaccadd命令の組み合わせ命令をld0add命令とし，ldata2命令とaccadd命令の組み合わせ命令をld2add命令とします．両方の命令はロード命令であるため，LSCXの領域を使用します．

```
opcode ld0add op2=4'b0010 LSCX
opcode ld2add op2=4'b0011 LSCX
```

iclassの定義は，次のようになります．

```
iclass l0a {ld0add} {inout ars}
          {inout acc32, in rdata, out data0}
{out VAddr, in MemDataIn16}
iclass l2a {ld2add} {inout ars}
          {inout acc32, in rdata, out data2}
{out VAddr, in MemDataIn16}
```

命令機能は**リスト26**のようになります．scheduleは**リスト27**のようになります．

次にldata1命令とmultie命令の組み合わせ命令では乗算ユニットへの入力を考慮しなければなりません．data0とdata1の乗算とdata1とdata2の乗算です．それぞれの命令をld1ml0命令とld1ml2命令とします．

〔リスト26〕命令機能

```
reference ld0add {
        assign VAddr = ars;
        assign data0 =
                MemDataIn16;
        assign ars = ars + 2;

        assign acc32 = acc32 +
                        rdata;
}

reference ld2add {
        assign VAddr = ars;
        assign data2 =
                MemDataIn16;
        assign ars = ars + 2;

        assign acc32 = acc32 +
                        rdata;
}
```

〔リスト27〕スケジュール

```
schedule ld0a {ld0add} {
        def data0 2;
}

schedule ld2a {ld2add} {
        def data2 2;
}
```

〔リスト29〕data0とdata1へのロード命令とacc32にrdataを加算する命令

```
opcode ldata0 op2=4'b0000 LSCX
opcode ldata1 op2=4'b0001 LSCX

iclass ld0 {ldata0} {inout ars} {out data0} {out VAddr, in
                                              MemDataIn16}
iclass ld1 {ldata1} {inout ars} {out data1} {out VAddr, in
                                              MemDataIn16}

reference ldata0 {
        assign VAddr = ars;
        assign data0 = MemDataIn16;
        assign ars = ars + 2;
}

reference ldata1 {
        assign VAddr = ars;
        assign data1 = MemDataIn16;
        assign ars = ars + 2;
}

schedule ld0 {ldata0} {
        def data0 2;
}

schedule ld1 {ldata1} {
        def data1 2;
}

opcode addacc op2=4'b0010 CUST0

iclass ada {addacc} { } {inout acc32, in rdata}

reference addacc {
        assign acc32 = acc32 + rdata;
}
```

〔リスト28〕命令機能の内容

```
reference ld1ml0 {
        assign VAddr = ars;
        assign data1 = MemDataIn16;
        assign ars = ars + 2;

        wire s = 1'b0;
        wire [31:0] sum0 = TIEmul(data0, data1, s);
        assign rdata = sum0;
}

reference ld1ml2 {
        assign VAddr = ars;
        assign data1 = MemDataIn16;
        assign ars = ars + 2;

        wire s = 1'b0;
        wire [31:0] sum0 = TIEmul(data2, data1, s);
        assign rdata = sum0;
}
```

〔リスト30〕semantic

```
semantic gp1a {ldata0, ldata1, ld0add, ld2add, addacc}
{
        assign VAddr = ars;
        assign data0 = MemDataIn16;
        assign data1 = MemDataIn16;
        assign data2 = MemDataIn16;
        assign ars = ars + 2;

        assign acc32 = acc32 + rdata;
}

semantic gp1ml {ld1ml0, ld1ml2}
{
        assign VAddr = ars;
        assign data1 = MemDataIn16;
        assign ars = ars + 2;

        wire s = 1'b0;
        wire [31:0] d0 = (ld1ml0) ? data0 : data2;
        wire [31:0] d1 = data1;
        wire [31:0] sum0 = TIEmul(d0, d1, s);
        assign rdata = sum0;
}
```

〔リスト31〕Cソースコード

```
int fir tie3(x, y)
        short *x;
        short *y;
{
        int i;

        WPDATA(0);
        WACC32(0);

        /* Calculation of FIR
                    filter */
        ldata0(y);
        ldata1(x);
        for(i=0;i<128;i++)
        {
            ld2add(y);
            ld1ml0(x);
            ld0add(y);
            ld1ml2(x);
        }
        addacc();

        return(RACC32());
}
```

```
  opcode ld1ml0 op2=4'b0100 LSCX
  opcode ld1ml2 op2=4'b0101 LSCX
```

iclass は次のように定義できます.

```
  iclass lm0 {ld1ml0} {inout ars}
             {out rdata, inout data1, in data0}
  {out VAddr, in MemDataIn16}
  iclass lm2 {ld1ml2} {inout ars}
             {out rdata, inout data1, in data2}
  {out VAddr, in MemDataIn16}
```

命令機能の内容は,**リスト 28** のようになります.

schedule の定義は次のようになります.

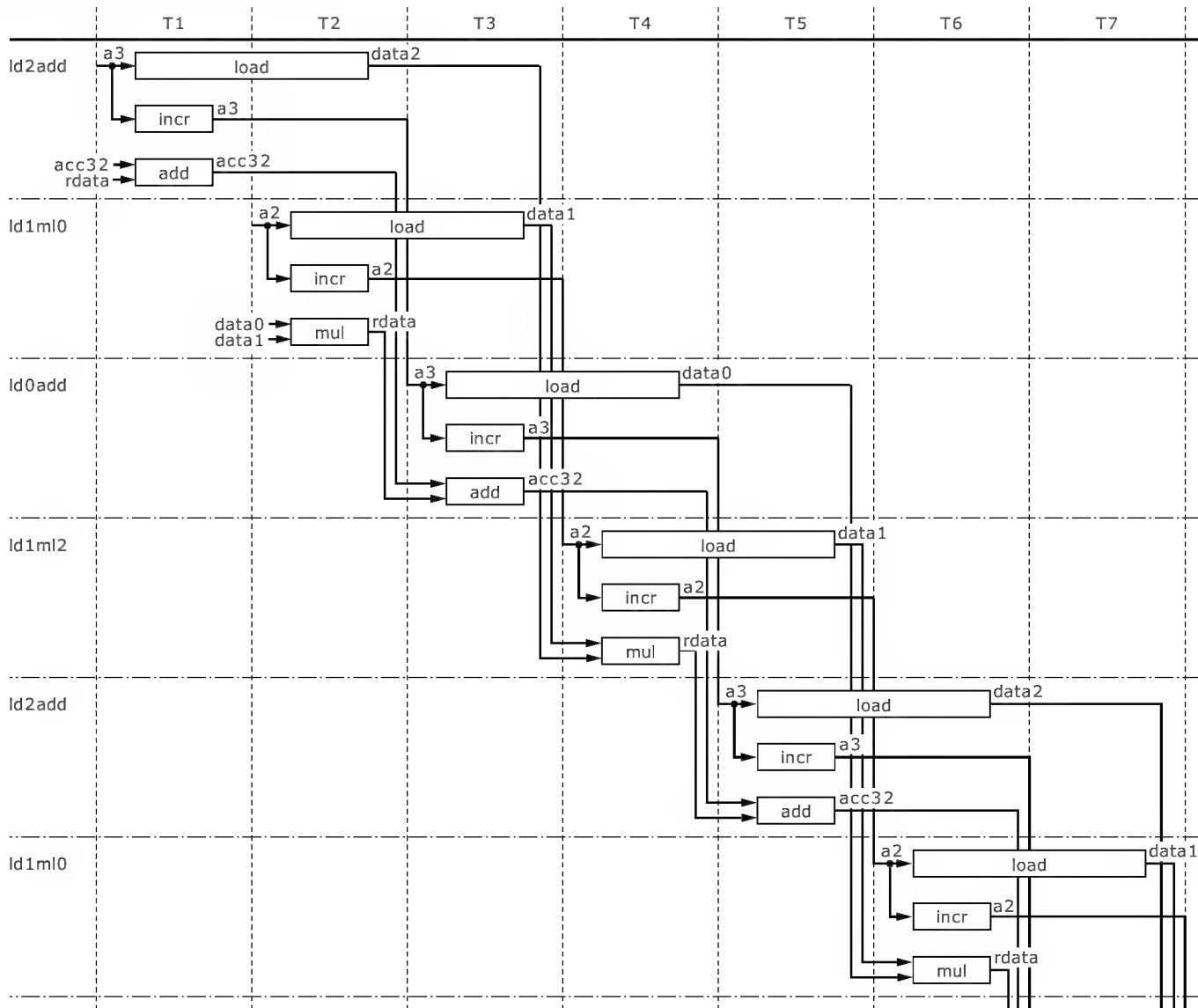
```
  schedule ld1 {ld1ml0, ld1ml2} {
  def data1 2;
  }
```

さらに,data0 と data1 へのロード命令と acc32 に rdata を加算する命令が必要となります(**リスト 29**).

以上で命令が設計できました.しかしこのままでは乗算器とアキュムレータへの32ビット加算器が2個ずつ必要になってしまいます.ところが,TIE 言語にはハードウェアリソースを共有することができる semantic という section があります.これを使用すると命令機能を reference の代わりとして,**リスト 30** のように semantic を記述できます.

これでアキュムレータへの32ビット加算器と乗算器は一つになりました.以上で三つの機能パイプライン命令の設計ができました.C ソースコードは**リスト 31** のようになります.

ループ内では二つの積和演算が実行されています.一方,ループに入る際の最初の ld2add 命令ではアキュムレータへの加算命令は機能しません.そこでループ後に addacc 命令を実行しなければなりません.**リスト 32** のようなアセンブラコードに展開されます.

〔図 12〕パイプラインデータフロー動作(5)

〔リスト32〕アセンブラコード

```
1  <fir tie3>:                   10          ld2add a3
2          entry a1, 32          11          ld1ml0 a2
3          movi a4, 0            12          ld0add a3
4          wpdata a4             13          ld1ml2 a2
5          wacc32 a4             14  L1:
6          ldata0 a3             15          addacc
7          ldata1 a2             16          racc32 a2
8          movi a4, 128          17          retw.n
9          loopnez a4, L1
```

〔リスト34〕iclassの定義

```
iclass l2m {ld2mla} {inout ars}
           {inout data2, inout acc32, inout rdata, in data3}
           {out VAddr, in MemDataIn32}
iclass l3m {ld3mha} {inout ars}
           {out data3, inout acc32, inout rdata, in data0,
                                                  in data1}
           {out VAddr, in MemDataIn32}
iclass l0m {ld0mla} {inout ars}
           {inout data0, inout acc32, inout rdata, in data1}
           {out VAddr, in MemDataIn32}
iclass l1m {ld1mha} {inout ars}
           {out data1, inout acc32, inout rdata, in data2,
                                                  in data3}
           {out VAddr, in MemDataIn32}
```

実際に，**図12**(前頁)のデータフローをもとにして何サイクルで積和演算が実行されているか確認します．ループは**リスト32**の10行目の`ld2add`命令から13行目の`ld1ml2`命令までの4命令です．T1で`ld2add`命令を実行します．この命令はアドレスレジスタa3でポイントするデータを`data2`にロードし，アドレスのアップデートを行い，アキュムレータ`acc32`に乗算結果を保持する`rdata`を加算する命令です．アドレスのアップデートはT1で更新されますが，`data2`へのロードは次のサイクルで実行されます．ループに入ったときの最初の`ld2add`命令では，`acc32`と`rdata`が初期化されて0となるため加算機能はダミー処理となります．もちろん，2回目以降の`ld2add`命令では加算機能が実行されます．したがって，ループ終了後に`addacc`命令が1度実行されます．T2では`ld1ml0`命令が実行されます．この命令はアドレスレジスタa2でポイントするデータを`data1`にロードし，アドレスのアップデートを行い，`data0`と`data1`の値を乗算し，結果を`rdata`に出力させる命令です．乗算はT2

〔図13〕4機能のパイプライン化の実行フロー

〔リスト33〕data2とdata3の領域を定義

```
state data0 32
state data1 32
state data2 32
state data3 32
state rdata 32
state acc32 32

user_register DATA0 0 {data0[31:0]}
user_register DATA1 1 {data1[31:0]}
user_register DATA2 2 {data2[31:0]}
user_register DATA3 3 {data3[31:0]}
user_register RDATA 4 {rdata[31:0]}
user_register ACC32 5 {acc32[31:0]}
```

で実行されますが，`data1`へのロードはT3で実行されます．つまり乗算時には`data1`の更新前の値を使用し，命令実行後に`data1`が更新される動作になります．そのため，`data0`と`data1`の値はこの命令が実行される前にロードしなければなりません．T3では`ld0add`命令を実行します．この命令はアドレスレジスタa3でポイントするデータを`data0`にロードし，アドレスのアップデートを行い，アキュムレータ`acc32`に前の命令で実行された乗算結果を加算する命令です．

T4では`ld1ml2`命令を実行します．この命令はアドレスレジスタa2でポイントするデータをdata1にロードし，アドレスのアップデートを行い，`data0`と`data1`の16ビット乗算を実行します．`data0`はT2でロードが完了し，`data1`はT3でロードが完了しています．つまり`ld1ml2`命令を実行する時には乗算に必要な二つのデータがすでにそろっていることになります．したがって，命令がstallすることはありません．T4でループが完了しT5では再度`ld2add`命令が実行されます．T6では`ld1ml0`命令が実行され，乗算機能では`data0`と`data1`を乗算します．ここでも`ld1ml0`命令を実行する際に`data0`と`data1`のロードは完了しています．以上のことから，4サイクルで二つの積和演算が，つまり**2サイクルで一つの積和演算が実行される**ことになります．

● 4機能のパイプライン化

積和演算を1サイクルで実行する際に必要な処理は，16ビット乗数と16ビット被乗数のロードと乗算と加算を機能パイプライン的に実行しなければならないことです．XtensaはRISCであることから，二つの異なるメモリ領域から二つのデータをロードすることはできません．そこで視点を変えて連続する二つの16ビットデータのロード，乗算，アキュムレータへの加算を1命令で実行することで1サイクルで一つの積和演算を実行することを考えます．

演算のフローは**図13**のように`ldata0`で32ビットの連続したデータをロードし，`ldata1`で連続したデータをロードします．1サイクルstallし，`mulh`で上位16ビットの乗算を行い，`mull.add`で下位16ビットの乗算とアキュムレータの加算を実行します．次に`add`でアキュムレータへの加算を行います．この一連のフローにより連続した二つの16ビットデータの積和演算が二つ

〔リスト35〕命令機能

```
reference ld2mla {

        wire [15:0] d21 = data2[15:0];
        wire [15:0] d31 = data3[15:0];

        assign VAddr = ars;
        assign data2 = MemDataIn32;
        assign ars = ars + 4;

        wire [31:0] tmp = rdata;

        wire s = 1'b1;
        wire [31:0] sum0 = TIEmul(d21, d31,
                                          s);
        assign rdata = sum0;

        assign acc32 = acc32 + tmp;
}

reference ld3mha {
        assign VAddr = ars;
        assign data3 = MemDataIn32;
        assign ars = ars + 4;

        wire [15:0] d0h = data0[31:16];
        wire [15:0] d1h = data1[31:16];

        wire [31:0] tmp = rdata;

        wire s = 1'b1;
        wire [31:0] sum0 = TIEmul(d0h, d1h,
                                          s);
        assign rdata = sum0;

        assign acc32 = acc32 + tmp;
}

reference ld0mla {
        assign VAddr = ars;
        assign data0 = MemDataIn32;
        assign ars = ars + 4;

        wire [15:0] d01 = data0[15:0];
        wire [15:0] d11 = data1[15:0];

        wire [31:0] tmp = rdata;

        wire s = 1'b1;
        wire [31:0] sum0 = TIEmul(d01, d11,
                                          s);
        assign rdata = sum0;

        assign acc32 = acc32 + tmp;
}

reference ld1mha {
        assign VAddr = ars;
        assign data1 = MemDataIn32;
        assign ars = ars + 4;

        wire [15:0] d2h = data2[31:16];
        wire [15:0] d3h = data3[31:16];

        wire [31:0] tmp = rdata;

        wire s = 1'b1;
        wire [31:0] sum0 = TIEmul(d2h, d3h,
                                          s);
        assign rdata = sum0;

        assign acc32 = acc32 + tmp;
}
```

〔リスト36〕スケジュール

```
schedule ld0ma {ld0mla} {
        def data0 2;
}

schedule ld1ma {ld1mha} {
        def data1 2;
}

schedule ld2ma {ld2mla} {
        def data2 2;
}

schedule ld3ma {ld3mha} {
        def data3 2;
}
```

実行されます．しかしパイプライン機能的に実行することを考えると，次のデータをdata0とdata1にロードすることはできません．これは乗算を実行する前に必要なデータを上書きしてしまうからです．そこで新しいステートレジスタdata2とdata3の領域を定義します(**リスト33**)．

実際にどのような命令の組み合わせが必要か考えます．ldata0とmull.addの組み合わせ，ldata1とmulhとaddの組み合わせ，ldata2とmull.addの組み合わせ，ldata3とmulhとaddの組み合わせが必要になります．

インターフェース信号は32ビットのロード命令を設計することから次のようになります．

```
interface VAddr        32 core out
interface MemDataIn32 32 core in
```

ldata0命令とmull.add命令の組み合わせ命令をld0mla命令とし，ldata1命令とmulh命令とadd命令の組み合わせ命令をld1mha命令とします．ldata2命令とmull.add命令の組み合わせ命令をld2mla命令とし，ldata3命令とmulh命令とadd命令の組み合わせ命令をld3mha命令とします．四つの命令はロード命令であるため，LSCXの領域を使用します．

```
opcode ld2mla op2=4'b0010 LSCX
opcode ld3mha op2=4'b0011 LSCX
opcode ld0mla op2=4'b0100 LSCX
opcode ld1mha op2=4'b0101 LSCX
```

iclassの定義は**リスト34**のようになります．命令機能は**リスト35**のようになります．scheduleは**リスト36**のようになります．さらに，data0とdata1へのロード命令(**リスト37**)とacc32にrdataを加算する命令が必要となります．

〔リスト37〕data0とdata1へのロード命令

```
opcode ldata0 op2=4'b0000 LSCX
opcode ldata1 op2=4'b0001 LSCX

iclass ld0 {ldata0} {inout ars} {out data0} {out VAddr, in MemDataIn32}
iclass ld1 {ldata1} {inout ars} {out data1} {out VAddr, in MemDataIn32}

reference ldata0 {
        assign VAddr = ars;
        assign data0 = MemDataIn32;
        assign ars = ars + 4;
}

reference ldata1 {
        assign VAddr = ars;
        assign data1 = MemDataIn32;
        assign ars = ars + 4;
}

schedule ld0 {ldata0} {
        def data0 2;
}

schedule ld1 {ldata1} {
        def data1 2;
}
```

以上で命令が設計できました．しかしこのままでは乗算器とアキュムレータへの32ビット加算器が4個ずつ必要になってしまいます．したがって，3機能パイプライン化と同様にsemanticを使用して設計します(**リスト38**)．

これでアキュムレータへの32ビット加算器と乗算器は一つになりました．以上で四つの機能パイプライン命令の設計ができました．Cソースコードは**リスト39**(p.183)のようになります．

ループ内では四つの積和演算が実行されています．一方，ループに入る際の最初のld2mla命令とld3mha命令ではアキュムレータへの加算命令と積和演算は事実上機能しません．そこでループ後にmla命令とaddacc命令を実行しなければなりません．**リスト40**のようなアセンブラコードに展開されます．

〔リスト 38〕semantic を使用して設計

```
semantic gpldm {ld2mla, ld3mha, ld0mla, ld1mha, mulh,
                                        mulla, ldata0, ldata1, addacc} {
        assign VAddr = ars;
        assign data0 = MemDataIn32;
        assign data1 = MemDataIn32;
        assign data2 = MemDataIn32;
        assign data3 = MemDataIn32;
        assign ars = ars + 4;

        wire [15:0] d0 = (ld2mla) ? data2[15:0] :
              (ld3mha) ? data0[31:16]:
              (ld0mla) ? data0[15:0] :
              (ld1mha) ? data2[31:16]:
              (mulh) ? data0[31:16]: data0[15:0];
        wire [15:0] d1 = (ld2mla) ? data3[15:0] :
              (ld3mha) ? data1[31:16]:
              (ld0mla) ? data1[15:0] :
              (ld1mha) ? data3[31:16]:
              (mulh) ? data1[31:16] : data1[15:0];

        wire s = 1'b1;
        wire [31:0] sum0 = TIEmul(d0, d1, s);
        assign rdata = sum0;

        wire [31:0] tmp = rdata;
        assign acc32 = acc32 + tmp;
}
```

〔リスト 40〕アセンブラコード

```
1  <fir tie4>:                  10          ld2mla a3
2          entry a1, 32         11          ld3mha a2
3          movi a4, 0           12          ld0mla a3
4          wpdata a4            13          ld1mha a2
5          wacc32 a4            14  L1:
6          ldata0 a3            15          ld2mla a3
7          ldata1 a2            16          addacc
8          movi a4, 64          17          racc32 a2
9          loopnez a4, L1       18          retw
```

実際に，**図 14** のデータフローをもとにして何サイクルで積和演算が実行されているか確認します．ループは**リスト 40** の 10 行目の ld2mla 命令からから 13 行目の ld1mha 命令までです．T1 で ld2mla 命令を実行します．この命令は a3 のポイントするメモリからデータを data2 にロードし，a3 のアドレスをアップデートします．また，アキュムレータに rdata の結果を加算し，data2 の下位 16 ビットと data3 の下位 16 ビットを乗算します．T2 では ld3mha 命令を実行します．この命令は a2 のポイントするメモリからデータを data2 にロードし，a2 のアドレスをアップデートします．また，アキュムレータに rdata の結果を加算し，data0 の上位 16 ビットと data1 の上位 16 ビットを乗算します．ここでは data0 と data1 はすでにロードされています．T3 では ld0mla 命令を実行します．この命令は a3 のポイントするメモリからデータを data0 にロードし，a3 のアドレスをアップデートします．また，アキュムレータに rdata の結果を加算し，data0 の下位 16 ビットと data1 の下位 16 ビットを乗算します．data0 を使用する命令で，data0 のロードを実行しているために，stall は発生しません．

T4 では ld1mha 命令を実行します．この命令は a2 のポイントするメモリからデータを data2 にロードし，a2 のアドレスをアップデートします．また，アキュムレータに rdata の結果を加算し，data2 の上位 16 ビットと data3 の上位 16 ビットを乗算します．以上のことから，4 サイクルで四つの積和演算が実行されます．つまり **1 サイクルで一つの積和演算**が実行されることになります．

〔図 15〕FIR フィルタ積和演算の平均サイクル数

1.6 FIR フィルタ演算の性能解析

実際に Xtensa の命令セットシミュレータで実行し，出力結果を解析してみます．C のソースコードでは関数コールの形式で記述されているため，関数コールによるオーバヘッドを除いたサイクル数から計算した 1 積和演算あたりのサイクル数を表記したものを**図 15** に示します．

ここで横軸は FIR のタップ数を表し，縦軸はサイクル数を表します．また，Mac16 は 1.1 で確認した Mac16 オプションを追加した場合の 1TAP の積和演算に必要な平均サイクル数，tie(1) は 1.2 で設計した TIE 命令による平均サイクル数，tie(2) は 1.3 で設計したデータロードにアドレスインクリメント機能をもつ TIE 命令の場合による平均サイクル数，tie(3) は 1.4 で設計した 2 並列の SIMD 演算の TIE 命令の場合による平均サイクル数，tie(4) は 1.4 で設計した 4 並列の SIMD 演算の TIE 命令の場合による平均サイクル数，tie(5) は 1.5 で設計したデータロードにアドレスインクリメント機能と加算あるいは乗算機能をもつ IE 命令の場合による平均サイクル数，tie(6) は 1.5 で設計したデータロードにアドレスインクリメント機能，加算機能と乗算機能をもつ TIE 命令の場合による平均サイクル数のグラフです．

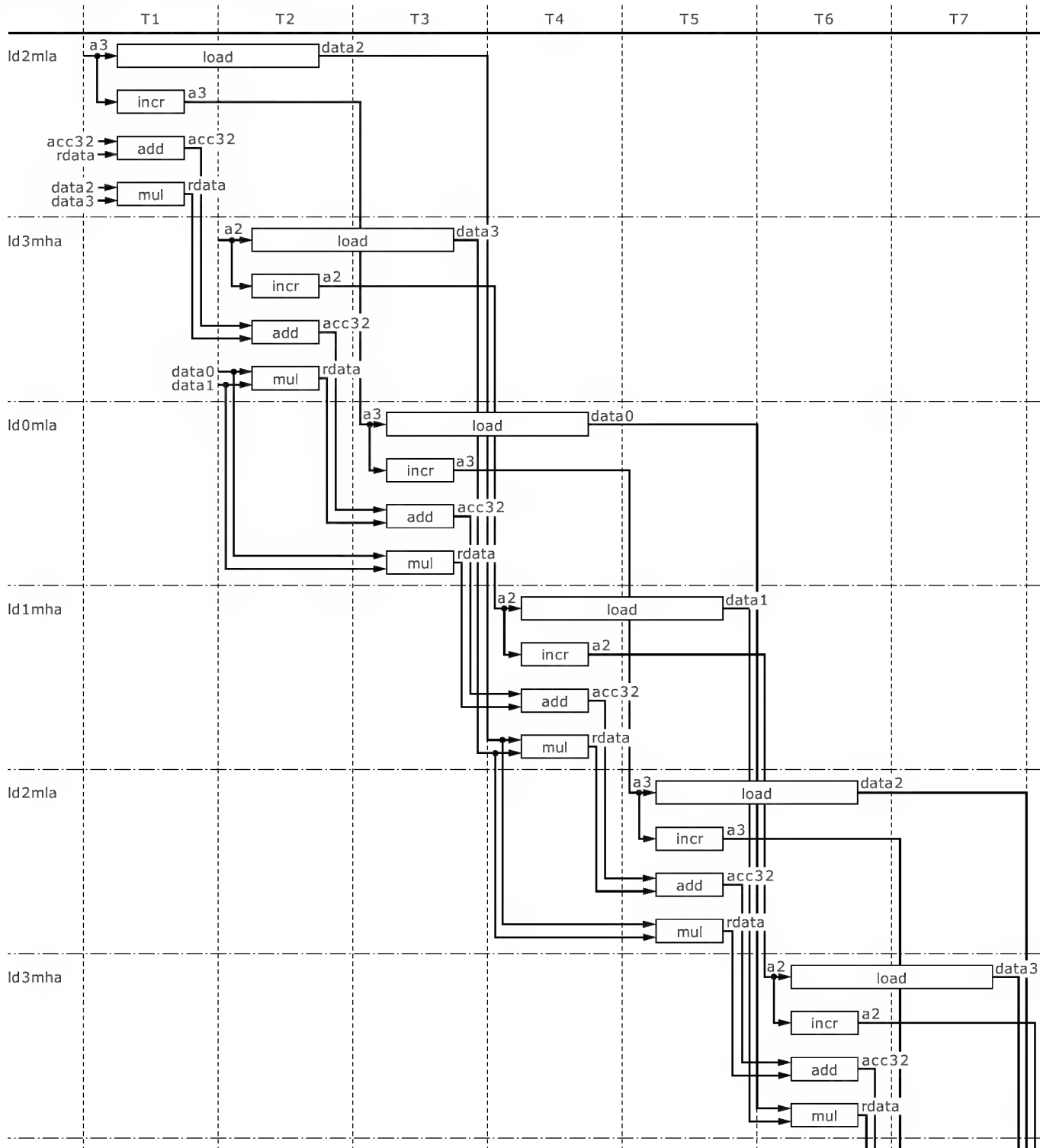
*　　　　　*

以上のように，SIMD 命令あるいはパイプライン化命令によって，プロセッサであるとはいえ，かなりの高速化を図ることができることをおわかりいただけたのではないでしょうか．本稿で紹介したのは，TIE 命令の可能性の一部でしかなく，アプリケーションによってはもっとさまざまな命令の組み合わせも考えられます．ユーザーによってそのアプローチはさまざまであり，またそれがコンフィギュラブルプロセッサとしての Xtensa の特徴でもあるので，ユーザーによって可能性を広げていくことのできる製品だといえます．

ながみね・じょう　テンシリカ(株)

〔リスト39〕Cソースコード

```
int fir tie4(x, y)
        short *x;
        short *y;
{
        int i;

        WPDATA(0);
        WACC32(0);

        /* Calculation of FIR
                        filter */
        ldata0(y);
        ldata1(x);
        for(i=0;i<64;i++)
        {
                ld2mla(y);
                ld3mha(x);
                ld0mla(y);
                ld1mha(x);
        }
        ld2mla(y);
        addacc();

        return(RACC32());
}
```

〔図14〕パイプラインデータフロー動作(6)

シニアエンジニアの技術草子 弐拾参之段

三文の得

旭 征佑

● カートリッジで生き残る

企業向けプリンタの売り上げの落ち込みは激しい．しかし，プリンタ市場の約85%を占めるのは，インクジェットプリンタだ．年間販売台数は700万台程度で伸び悩んでいるが，それでも，毎年，低価格化していく割には技術面や機能面での進歩が大きく，買い替え需要が期待できる．メーカーは，写真画質やフチなし印刷など，さらに高機能を訴えたコンシューマ製品で，売り上げ全体の底上げを図ろうとしている．と，ここまでが公式にいわれていることだ．

でも，それだけではない．メーカーには絶対にシェアを落としてはいけない理由があるのだ．なぜならば，シェアはインクカートリッジの販売量に直接関わってくるからだ．

インクカートリッジは，ただのインクをパッケージ化したにすぎない．大量生産すれば製造原価は数十円以下だろう．それを，千円以上で売るわけだからメーカーは大きく儲かるはずだ．じつはプリンタメーカーの利益の大半は，インクカートリッジが占めている．つまり，インクジェットプリンタのシェアを落とすことは，すなわち大きな利益を失うことを意味するのだ．

カートリッジの種類がやたらと多いのも，他業者の参入をしづらくする戦略の一つなのだろう．多種類のカートリッジを製造すると，最初に「型」を作るコストもかさむ．大量に販売できないと，とてもではないが採算が取れなくなる．

そんな割高に思えるカートリッジも，メーカーが良い製品を開発するコストを回収するためのものだろう．そう思えばある程度甘受できるかもしれない．しかし，ターゲットが一般コンシューマである限り，これ以上の製品の必要はないのでは，などと思うのは，筆者だけだろうか．

● そういえばコピー業界

古い友人に，最大手クラスのコピー機メーカーの営業マンがいる．20年ぐらい前の話だ．彼の仕事のメインは，古いコピー機を引き取り，新品のコピー機にリース契約させることだった．彼の事務所には，古いコピー機が山のように積み上げられていたらしい．筆者は，「欲しければいくらでもやるよ，みんな他社製品だけどね」といわれていた．

当時，コピー機は，ほぼ数社の寡占状態になっていた．製品の技術革新も速く，ユーザーにも買い換えるメリットがあったのだろう，メーカー間の激しいシェア争いが続いていた．

しかし，その友人がいうには，競争激化の本当の理由は別にあったという．当時の彼の言葉を思い出す．「トナーはドラム缶1本10円の世界，それを小分けして数万円で売れるわけだから，大きな儲けが出る．他社のシェアをどんどん食って，トナーで儲けたところが生き残る」．その戦略が効を奏したのか，彼の会社は間もなく業界のトップクラスに踊り出た．

その後，開発競争が一段落すると，コピーは価格競争に突入し，体力のないコピーメーカーの中には脱落していくところもあった．最近は，プリンタなどの機能を含んだ複合機やカラーコピーに，販売の主力が移ってきたようだ．一般のコピー機市場では，企業サイドのコスト意識の高まりもあり，今でも大幅な値引き合戦が繰り広げられている．

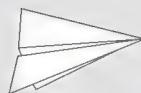
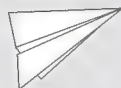
● 再生の術(パート1)

話をプリンタに戻そう．仕事ではもっぱらレーザープリンタだが，トナーが切れると，インクジェットでしばらくごまかしてしまうことがある．トナーは1本1万円以上するが，インクカートリッジは比較的安いので，急ぐときは近所で買うことも多い．1本千数百円はあたりまえなので，カラーインクと合わせて買うと3,000円を超えたりする．そんなときに限って，近くに1万円以下で売っている特売プリンタがあったりする．若干の矛盾を禁じえない．

あるとき，レーザープリンタが壊れた．修理が面倒だったので，しばらくインクジェットを使用していた．最近のプリンタドライバはインテリジェント化がすすみ，インクの残量が表示され，交換時期まで教えてくれる．

ちょうどインクがなくなったカートリッジがあったので，親指と人差し指で強く押してみた．すると中のほうからインクがブクブクと湧き出してくるのが見えた．これは，どうみてもまだまだインクが残っているではないか．

試しに，そのカートリッジを，再度プリンタにセットしてみた．セットが完了すると，なんとインク残量満タンの新品のインクカートリッジに化けてしまった．印刷すると，問題なく印刷ができる．なんだ，まだ使えるではないか！

そのカートリッジはけっこう長く使えたが，あるとき，突然インクが切れてしまった．インク残量表示を見たら，約半分を

きったところぐらいだろうか．都合，インクカートリッジが1.5倍使えるようになったことになる．インク残量を何で決めているのか不思議だったが，それはさておいて，なにか得をした気持ちになった．筆者は，これを「カートリッジ再生の術」と呼ぶことに決め，勝手にあちらこちらで吹聴している．

● 再生の術（パート2）

もう一つのインクジェットのインクも試してみた．しかし，こちらはインクがなくなったカートリッジを再セットすると，プリンタがエラーになり動かない．どうもカーリッジについているセンサが，「インクがない」と知らせているようだ．

そのカートリッジを分解してみた．カッターを使って上部の接合点に切り込みを入れると，意外と簡単に分解できた．中には目の細かいスポンジのようなものが詰まっていて，インクが染み込んでいるだけだ．センサは，よく見たら電極しかない．どうやらカートリッジ内の電気抵抗を測ってインクの残量をみているのだろう．そのセンサをよく見ようと，カートリッジを横に倒すと，今度はインクがにじみ出てきた．さらに，カートリッジの両端を指で軽く押すと，大量のインクがこぼれるほど湧き出してきた．これも再生できそうだ．

興味半分，しばらくこのインクジェットを使うことにした．カートリッジは厳重にテープでとめて保存し，2本たまった段階で解体し，インクを片方に移した．ちょっと多かったかな，と思いつつもカートリッジのふたを瞬間接着剤で丁寧につなげて再生させることができた．現在も使用しているが，まだこのインクはなくなっていない．しかし，溢れ出たぐらいだから，きっと1個分以上は使えるのだろう．

分解したり接着する手間もかかるし，どうしても手や周りがインクで汚れるので，差し引きたいして得ではないのかもしれない．まあ，気持ちだけお得というか……．

● 来るか，価格破壊

プリンタの市場はエプソンとキヤノンがしのぎを削り，2社で80%にも迫る寡占状態が続いている（ガートナーグループ調査）．これも，インクカートリッジが割高な理由の一つかもしれない．しかし，プリンタの性能も一段落した．以前のコピー機と似たような状況が，もしかしたら迫ってきているのかもしれない．あとは，もう少し，ユーザーがコスト意識に目覚める必要があるだろう．「なぜ，インクカートリッジはこうも高いのか！」と．

9月24日にDELLがプリンタ専業メーカーのLexmarkと契約した．パソコンのシェアを一気に15%まで伸ばした直販メーカーが，プリンタ市場に殴り込みかけることは，たいへん興味深い．日本人は同じメーカーの製品をセットで買いそろえる傾向があるからだ．もちろん，DELLはインクカートリッジも販売する．DELLが，サプライ品をどんな手段を使って供給するか，というのは，それはそれで興味の尽きないところだ．でも，この時期に参入するというのでは，やはりどうしても価格破壊を期待してしまう．DELLのがんばりに期待したい．

あさひ・しょうすけ　テクニカルライター
イラスト：森　祐子

注：どのメーカーのプリンタでも使用済みカートリッジが再利用できるとは限りません．また，万一プリンタが壊れても補償できませんので，試す場合には自己責任で行ってください．

Engineering Life in Silicon Valley 対談編

インターネットバブルの前と後の比較

H. Tony Chin

> 今回のゲストのプロフィール
> 吉田　穣(よしだ・ゆたか)：1963年愛知県生まれのIC設計エンジニア．1988年東京大学相関理化学修士卒．10年間日本でIC設計経験を積んだのちに渡米し，1999年からシリコンバレーでさまざまなスタートアップでIC設計を行う．趣味は温泉とスキー．

☆ 10年前のシリコンバレーのイメージ

トニー まずは，シリコンバレーに来た経緯から話を進めていただけませんか？

吉田 日本で入社した会社がシリコンバレーのVLSI Technology, Inc.(以下VLSI，現在はPhilipsの一部)と提携していました．この関係でVLSIのボストンにあるデザインセンタに約3か月，シリコンバレーではサンノゼのデザインセンタに約8か月滞在しました．こちらの設計環境を学んだり顔つなぎがおもな目的でした．

そこではARM6と7の改良版の作成に参加しました．具体的な内容は，フルカスタムですでにできていたチップをVLSIのスタンダードセルに置き換えてパフォーマンスを上げることです．結局，できあがったデバイスは20～30％のパフォーマンスアップが達成できて，フルカスタム版と見劣りしない出来でした．

その当時のアメリカ人の上司であるアンドレは，非常に喜んでくれ，そのままアメリカに残らないか？と聞かれました．

トニー 短い期間で結果を出されたのですから，こちらの設計担当マネージャも欲しがるでしょうね．それで10年前ほどのシリコンバレーはどうでした？　90年ぐらいですよね？

吉田 当時はシリコンバレーのイメージが非常に良かったので，ぜひ仕事をしてみたいと思いました．皆スマートに仕事をしているし，環境も整っているというイメージです．まわりにいたエンジニアは優秀な人が多かったですね．また，家賃は今のように高くなかったし，高速道路も比較的空いていました．家賃の高騰と渋滞はのちのインターネットバブルのおかげですが(笑)．それから，当時は私の興味のある，面白そうなプロセッサ関係の会社がたくさんあったことも理由の一つです．

ほかには，プライベート面でもすごく楽しいんじゃないか？というイメージがありました．私は1人で渡米したのですが，上司が非常に気をかけてくれました．彼の夫婦は子供がなく，よく奥様の家族が所有するハウスボートとかに招待されました．シャスタ湖にハウスボートがあったので，たしか奥様のお父様がジェットスキーやバイクとかボートのディーラーとかやっていて，新製品が出たら試乗させくれるとか……アメリカでもなかなか少ないと思うのですが，一緒に楽しませてもらいました．

吉田　穣氏

トニー シャスタ湖はとても綺麗なところですよね．そこでウォータースポーツをするのは最高でしょう！

吉田 そうですね．結局私は日本にある会社の研修で渡米している立場上，そのまま「アメリカに残ります！」では済まないと思い，とりあえず一度帰国したのです．その後，その当時の上司は違う会社……日本でも少し有名になった，グラフィックス関係会社であるChromatic Researchに行ってしまいました．

しかし，その後も同じ元上司から声がかかりました．私は，日本に帰国してからはプロジェクトリーダーやマネージャをやっていたので，なかなか手が空かない状態でした．またその間にChromaticが駄目になり，元上司はどこかまた違う会社に行ってしまいました(笑)．

トニー う～ん，なかなかシリコンバレーらしい話ですね(笑)．

☆ インターネットバブルの絶好調でイメージダウン?!

吉田 結局　99年あたりに渡米してこの元上司を窓口に面接を受けたのですが，英語でワーっと言われただけで話せませんでした．まったく英会話の練習もしてなかったし，研修で来ていたときはなんとか生活できたので大丈夫だろうと思っていました……でも結局喋れなかったですね．これで面接には見事落されてしまい，かなりへこんでしまいました(笑)．

それでまた日本に戻って仕事となりましたが，シリコンバレーに戻りたいという気持ちが出てきて，雑誌の求人広告でシリコンバレーの会社があったので応募してみました．この会社は日本人が多い会社で，上層部の人が来日するということがあったので，日本での面接になり，結局ここに転職することが決まりました．そして　99年の10月あたりにまたシリコンバレーに戻りました．

トニー 今回のインターネットバブルが絶好調の頃ですよね？

吉田 そうです．しかし今度はすごいショックでした．家賃は2倍以上になっていたし，どの道路も車で渋滞が多く，乱暴に運転する人が増え……もっともショックだったのはエンジニアの質が落ちたような気がしたことです．

トニー そうですよね，私も家賃とか生活面での変化には非常に驚きました．では，そのエンジニアの質について具体的に説明していただけませんか？

吉田 たとえばコントラクタ(個人または請負設計会社から

のエンジニア)のことでしょうか……日本人のエンジニアに混じって多くの中国人系やインド人系のエンジニアが働いていました．大量に仕事を抱えていたので，社内のエンジニアだけでは足りないことが多く外部からコントラクタを使っていました．

VLSIでもコントラクタを使っていたので一緒に仕事することがあったのですが，以前の記憶ですと，コンストラクタというと，しっかり仕事をしてくれる頼もしいエンジニア……職人肌系のエンジニアですよね，そういうイメージがありました．

それが，インターネットバブルの頃だと，ろくに仕事をしないのに＄250近い時給が支払われていました．小遣い稼ぎでやっているというタイプがほとんどでした．まあ，たまたま雇ったコントラクタがそうだったのかもしれませんが，昔のタイプの人と遭遇することはありませんでした．笑ってしまったのがツールの使い方を知らないということで，ツールの勉強をして，そのまま何もしないで辞めてしまう人がいたことです．

トニー 私もコントラクタというと「必殺仕事人」みたいなイメージで，本当に困ったときに助けてもらう，プロ中のプロのエンジニアという印象があります．だから社内で難しいアナログやRFの部分をやってもらうこととかが多かったです．しかし，お話だと本当にひどいですよね．

吉田 まあ，質の悪いコントラクタももちろん問題なのですが，雇うほうも問題ですよね．この会社では，やはりいくらシリコンバレーといえども，レベルの低いエンジニアやマネージャもいるということを再認識しました．エンジニアやマネージャの数が多いので，レベルのすごく高い人もいれば低い人もいるということです．

☆ 英会話はやはり難しい?!

トニー 現在勤めている会社は，通信とかの関連でかなり特別なデバイスをやってますよね？

吉田 動作スピードも速く，2.5GHzで動くI/Oをもっていて，こういうところに惹かれて入社しました．最近多いタイプの会社で，ファブレスで台湾の半導体ファブに実際のチップの製造をしてもらいますが，そのほかはわれわれの会社ですべて用意します．ディジアナのエキスパートも社内に抱えています．入社したきっかけは，また以前の上司でした．今回はシリコンバレーの仕事でだいぶ英会話も上達したので…….

トニー 日本人が多い会社ではなかったのですか？

吉田 そうですが，会議などはほかのエンジニアがわかるように英語でした．まあ，まだまだ得意ではありませんが，英会話学校に行く余裕もありません．現在の会社ではたまにスピーカホンなどで会議があるのですが，これは冷や汗物です．メールなど，書き物になると問題ないのですが…….まあ，ほかの外国人も多いので，わからないならわからないなりに皆意味を理解しようとしてくれるし，うまい具合に解釈してもらっているのでありがたく思っています．たまに，びっくりするほど私の言いたいことをちゃんと理解してくれることがあります．

☆ レイオフを経験する

トニー テロ後に一気にバブルが弾けた状態ですが，レイオフはありましたか？

トニー・チン氏

吉田 ありました．やはりテロ後に受注していた発注がキャンセルをしたりして，コストカット分を会社側がすることになり，それで人員削減が行われました．ちょうど11月頃に私は担当していた設計のテープアウト間近で仕事をしていました．それどころではなかったので，社内発表を聞いても，「こんなに忙しい時期なのにレフオフなんかバカバカしい，やれるもんならやってみろ！」とイライラしたぐらいでした．結局，11月の半ばに数人の上司に呼ばれたのですが，じつは呼ばれたのは残る人達でした．110名の中から20名強ぐらいがレイオフの対象になりました．私の設計グループでレイオフの対象になる人のコンピュータのアカウントが，すでにブロッキングされていてサボタージュができないようになっているということも聞かされました．

トニー レイオフされる人のリストを聞いてどうでした？

吉田 まあ，とくに驚きませんでした．妥当な線でした．でも疑問に残った人もいました．会社側からは，すでにプロジェクトで仕事がほぼ終わっている人から優先して切ったと聞かされたのですが，なんとなくインド系のエンジニアがたくさん残っているような気もしました．偶然かもしれませんが，経営陣がインド系だからなのかと勘ぐったりもしました．もっとあまり仕事ができない人を切れば良いのにとも思ったのですが…….

トニー 仕事には何か影響がありましたか？

吉田 もうパニック状態でした．サボタージュのほうに気が行ってたのか，まったく引き継ぎがありませんでした．テープアウトの直前で，検証系の仕事が多かったのですが，その人達が大量に辞めたので誰もいない状態でした．また引き継ぎがなかったので，どれが何かわからない！という状態でしたね．結局は辞めた人のディレクトリなどを調べてなんとかしましたが，テープアウトは1か月遅れ，ほぼ毎日徹夜状態が長く続きました．

☆ 頑張れ日本のエンジニア！

トニー まだまだ景気が回復しませんが……今後は？

吉田 仕事内容が面白いのでがんばる予定ですが，最近はまだまだ日本のエンジニアや会社も競争力があると感じています．日本のハードウェアのエンジニアの，細かい作業が得意なところや責任をもって仕事をする姿勢などはまだまだ競争力があると思います．ですから日本に戻って仕事もしてみたいと思っています．たとえば私の会社のような通信の分野で，日本の会社があまり活躍していないのが少し寂しいですから…….

対談を終えて：

多くの日本からのエンジニアと話す機会がある．今回の対談では，シリコンバレーがなにか非常に良い環境であるというイメージからもっと踏み込んでいたと感じた．吉田氏はシリコンバレーの良いことも悪いことも冷静にとらえており，非常にフランクで興味深い対談だった．

トニー・チン　htchin@attglobal.net　WinHawk Consulting

HARD WARE

●組み込み向けマイコン

SH7300

- 高速データ通信をベースとした次世代携帯電話のアプリケーションであるTV電話などの動画像や音声処理を高速に実行する，SH-3 DSPコアのCPU.
- MPEG-4の処理を高速に行えるエンコード/デコード処理用のハードウェアアクセラレータを搭載．ハードウェア処理により低消費電力化を図り，CPUの負荷を1/5に軽減することで，MPEG-4の処理性能を2倍以上に向上.
- SXGAサイズのカメラを直接接続できるインターフェースを内蔵．高精細カメラの大容量画像データを高速に取り込むことができるため，画像処理の高速化を実現．電子ズーム表示など高精細カメラ画像による多彩な表示が可能であり，TV電話などでスムーズで快適な表示を実現可能.

■ （株）日立製作所
価格：¥3,000（10,000個時）
TEL：03-5201-5234

●1チップマイコン

V850ES/SG2 V850ES/SJ2

- V850ES/SG2は100ピン，V850ES/SJ2は144ピン品.
- V850ESコアを搭載し，20MHzもしくは32MHz動作が可能.
- 従来品と比較して約25%の低消費電力化を実現.
- 640KバイトのROMおよび48KバイトのRAMの大容量メモリを搭載.
- ディジタル処理に適する3V単一電源と，従来のオーディオ機器用に普及していた5V系デバイスからの入力も可能なI/Oを提供することで，5V電源と3V電源が混在するシステムにも対応.

■ 日本電気（株）
サンプル価格：V850ES/SG2 ¥2,500～¥2,700
V850ES/SJ2 ¥3,000～¥3,400
TEL：044-435-9494 FAX：044-435-9608
E-mail：info@lsi.nec.co.jp

●1チップマイコン

SH7058F

- 最高動作周波数は従来品の2倍である80MHzで，104MIPSの処理性能を実現した，SH-2コアのCPU.
- 1Mバイト大容量フラッシュメモリは，80MHzでの1サイクルアクセス動作が可能.
- －40℃～125℃の広い温度範囲において動作し，エンジン近傍への制御機器の設置など，厳しい使用環境に対応可能.
- フラッシュメモリへの書き込み/消去プログラムを内蔵.
- フラッシュメモリのモードとして，ユーザーブートモードを追加．機器の電源投入後のブート動作プログラムを書き込むことが可能.
- 発振停止検出機能を追加することで，イネーブル時にはLSIの発振停止や異常を検出時に自動的に自己発振を開始.

■ （株）日立製作所
価格：¥6,000（1,000個時）
TEL：03-5201-5218

●車載向け16ビットマイコン

H8S/2282F

- スピードメータ，タコメータ，燃料計，水温計の4ゲージを駆動するステッピングモータを直接接続可能なドライバ内蔵モータコントロールPWMタイマを搭載.
- 走行距離を表示するための液晶表示を28セグメント×4コモンまで直接駆動可能.
- Bosch CAN Ver2.0B active規格に準拠したHCANを内蔵．CANインターフェースのデータバッファは16メッセージが格納可能で，最大通信速度1Mbpsを実現.
- 低消費電流モードからの復帰用に，CANのバス動作でマイコンを起動するウェイクアップ機能を内蔵.
- 100ピンQFPの小ピンコンパクトパッケージを実現.

■ （株）日立製作所
サンプル価格：¥1,700
TEL：03-5201-5212

●ディジタルテレビ用チップセット

LC74152B/LC74186E

- LC74152Bはオーディオ/ビデオデコーダLSIで，トランスポートストリーム，デマルチプレクサとMULTI2方式デスクランブラ，OSDコントローラ，ビデオスケーラなど，CPU以外のディジタルTVの受信に必要な機能を内蔵している.
- ビデオデコーダ部はBSディジタル放送をはじめとする，さまざまなディジタル放送の映像フォーマットに対応し，SDTV解像度でデコード出力を行う.
- オーディオデコーダはMPEG-ACCやMPEG-BCの音声符号化方式に対応している.
- LC74186EはPowerPCコアのディジタルTVシステムコントロール用マイコン.

■ 三洋電機（株）
サンプル価格：¥8,000（LC74152B）
¥4,500（LC74186E）
TEL：03-3837-6345 FAX：03-3837-6378

●ディジタルAV機器向けCODEC LSI

MB86392/MB86394

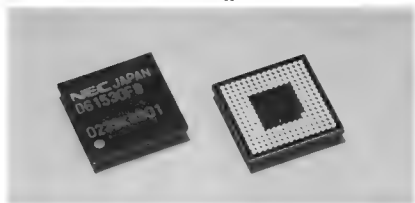
- MPEG-2準拠のディジタルAV機器向けCODEC LSI.
- ファームウェアダウンロード方式を採用しており，対応するファームウェアを変更することで，MPEG-1/2のほかに，MPEG-4やMPEG-ACC，MP3などのさまざまなオーディオフォーマットへの対応が可能.
- MB86392は，音声，画像対応メモリ搭載型MPEG-2AV CODEC LSIで，64MビットのFCRAMをSiP技術により同一パッケージ内に搭載．携帯AV機器向けにサイズ16×16mmの省スペース，標準800mWの低消費電力を実現.
- MB86394は，外付けメモリを最大512Mビットまで拡張することができるため汎用性が高い.

■ 富士通（株）
サンプル価格：¥6,000（MB86392）
¥4,000（MB86394）
TEL：042-532-2135
E-mail：edevice@fujitsu.com

●OFDM復調LSI

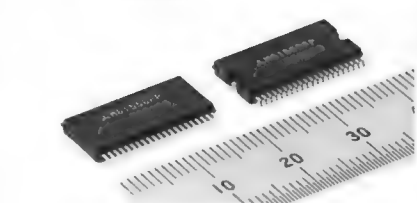
μPD61530

- 国内地上波ディジタル放送規格に準拠したOFDM復調回路，誤り訂正回路など地上波ディジタル放送受信機に必要な機能を1チップ上に集積したLSI.
- 独自の回路アルゴリズムの開発により，世界最小の40mWという低消費電力化を実現.
- チップサイズ9×9mmを実現.
- 電源電圧使用範囲が2.7～3.6Vと広範囲で，基準クロック周波数の選択範囲が2.2MHz～4.6MHzの任意の周波数に対応.
- トランスポートストリーム出力は，パラレル/シリアル選択が可能.

■ 日本電気（株）
サンプル価格：¥3,000
TEL：044-435-9494　FAX：044-435-9608
E-mail：info@lsi.nec.co.jp

●ディジタルパワーアンプ用ドライバLSI

M61556FP/M61558FP

- AVアンプ，DVDレシーバ，TVなどのオーディオアンプをフルディジタル化するディジタルパワーアンプ用ドライバLSI.
- M61556FPは100W×1チャネル出力のプリドライバLSI，M61558FPは30W×1チャネル出力のパワードライバLSI.
- M61556FPは，NチャネルMOS FETと組み合わせることで，1チャネルあたり100W（8Ω）の大出力を実現．中高級AVアンプに適する．デッドタイム調整回路，ブートストラップ用ダイオード，減電圧検出回路，クロックロス検出回路，温度検出プロテクタ用コンパレータなどの回路および素子を内蔵しているため，周辺回路を削減したコンパクトな設計が可能.

■ 三菱電機（株）
サンプル価格：¥600
TEL：03-3218-9450

●ファーストサーチ監視コプロセッサ

Vichara81000

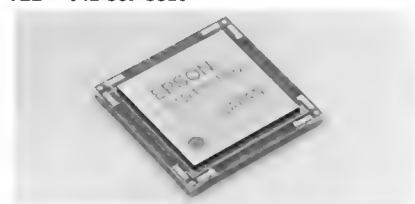
- ネットワーク機器で必要となるパケット操作の検索関連の機能を高速化するコプロセッサ.
- ASICとNPUによる検索命令シーケンスと関連パケットデータの負担を全範囲で軽減し，レイヤ3～7までのマルチ検索を管理するパケットコプロセッサ.
- 266MHzで動作し，複数の検索インデックスのサイプレスネットワークサーチエンジンと関連データのNoBL SRAMをあわせて使うことで検索操作を管理する.
- 最大四つのパケットプロセッサについてパラレルで複数のコンテキストをインターリーブおよび実行することで，四つそれぞれのポートで最大64のコンテキストをサポートする.
- Intel，AMCCおよびIBMのNPUを含むLA-1準拠NPUは，コプロセッサによる検索負荷の軽減によりLA-1バスへの変換が高効率で行われる.

■ 日本サイプレス（株）
サンプル価格：¥31,250（1,000個時）
TEL：03-5371-1921　FAX：03-5371-1955

●LCDコントロールLSI

S1D13710/S1D13712

- 従来のLCDコントロール機能に加え，ハードウェアJPEG回路，カメラインターフェース，メインとサブの2LCDインターフェースを内蔵.
- 画像表示や写真データ処理などの大きな負荷を受け持ち，高速処理を行うことが可能.
- ベースバンドCPUの負荷を大幅に軽減することが可能.
- アプリケーションプロセッサを採用する場合と比較しても，LCDコントローラはハードウェアで構成されているため，同じ動作周波数でも高速応答，低省電力化を実現可能.
- S1D13710は16ビット80系CPUインターフェースのみ，S1D13712は加えて68系CPUインターフェースをもつ.

■ セイコーエプソン（株）
サンプル価格：¥1,200
TEL：042-587-5816

●オンチップシステムソリューション

CC1010

- ノルウェーのChipconAS社が開発した，オンチップシステムシングルチップRFトランシーバIC.
- 300MHz～1000MHz周波数範囲での使用を可能にしたCMOS CC1000と業界標準8051マイクロコントローラコアを統合.
- 独自のSmartRF02テクノロジをベースにCMOS 0.35μmプロセスで開発.
- 32Kバイトインシステムプログラム化可能フラッシュ，ハードウェアDES暗号化/暗号解読および3チャネル10ビットA-Dコンバータ機能をもつ8051対応マイクロコントローラ内蔵.
- 315，433，868および915MHzのISM/SRDバンドのFSKシステムにデザイン.

■ テクセル（株）
サンプル価格：¥740（5,000個時）
TEL：03-5467-9273
E-mail：chipcon@teksel.co.jp
URL：http://www.teksel.com/

●ディジタルオーディオレシーバ

CS8416

- CS8416は，最大192kHzのサンプルレートでディジタルオーディオデータを受信/復号化し，低ジッタのクロックリカバリメカニズムを利用して，入力したオーディオストリームからクリーンなクロック信号に復元．8:2の入力マルチプレクサにより8種のディジタルオーディオソースの入力が可能.
- マルチプレクサの第2出力は，SPDIFパススルー機能を提供するためシステムの柔軟性が向上．圧縮されたオーディオ入力ストリームの自動検出とCD-Qサブコードの復号機能を備えており，3本の汎用出力ピンのいずれかを選択して信号を出力することが可能.

■ シーラス・ロジック（株）
価格：CS8416 $2.56（10,000個時）
TEL：03-5226-7378　FAX：03-5226-7677

HARD WARE

●フラッシュメモリ用コントローラ

GBDriver RA2/ GBDriver SA3

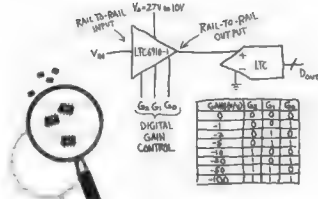
- NAND型フラッシュメモリの制御用コントローラ.
- GBDriver RA2は，フラッシュメモリ書き込み時のピーク電流を低く抑え(30mA，待機時200μA)ながら，各種のシステム，ユーザーデータ格納に十分な高速書き込み性能(1.5Mバイト/s)を実現．256Mバイトのフラッシュメモリを最大8個まで制御可能．ホストインターフェースは，PCMCIA ATA，コンパクトフラッシュおよびIDEインターフェースをサポート．
- GBDriver SA3は，SmartMediaを応用した各種システムに使用可能なIDEインターフェースおよびコンパクトフラッシュ，PCMCIA ATAインターフェース内蔵のフラッシュメモリコントローラ．256MバイトまでのSmart Media規格に対応する．

■ TDK(株)
サンプル価格：¥1,000(10,000個時)
TEL：03-5201-7102

●プログラマブルゲインアンプ

LTC6910-1

- 3ビット制御入力を使用して，0，1，2，5，10，20，50，100V/Vの反転利得をディジタルで選択可能．
- 8ピンSOT-23パッケージで供給され，サイズは9m㎡.
- 11MHzの利得帯域幅積，9nV/√Hzの入力基準ノイズ電圧密度，0.003%の低ひずみが特徴．
- レールトゥレール入力範囲，レールトゥレール出力振幅．
- 合計2.7V～10Vの単一または両電源で動作し，コマーシャル温度範囲とインダストリアル温度範囲の動作で完全に規格化．
- ダイナミックレンジは119dB.

■ リニアテクノロジー(株)
サンプル価格：¥108～(100,000個時)
TEL：03-5226-7291　FAX：03-5226-0268

●NOR型フラッシュメモリ

μPD29F256115 μPD29F256415

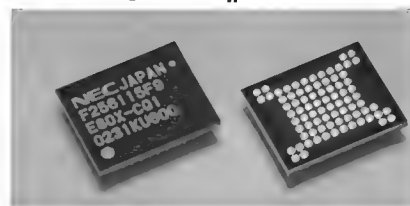
- 0.15μmの微細プロセスや回路配置の工夫などにより，容量256MビットのNOR型フラッシュメモリを実現．
- メモリセル部の面積を通常の1/2にしたことにより，単位チップ面積あたりのメモリ容量を増やし，低コスト化を実現．
- 読み出し回路などの工夫により，従来品64Mビットフラッシュメモリと同等の85nsという高速アクセスを実現．
- ページ内のデータを高速に出力するページアクセス機能を設けており，25nsという高速なページアクセス速度を実現．

■ 日本電気(株)
サンプル価格：¥5,000
TEL：044-435-9494　FAX：044-435-9608
E-mail：info@lsi.nec.co.jp

●PCインターフェースLSI

R5C592/R5C593

- PCカード，IEEE1394，小型フラッシュメモリカードの3種類のインターフェース機能を1チップに集積化したPCインターフェースLSI.
- R5C592は，PCカードインターフェースを2スロット，IEEE1394インターフェースを2ポート，小型フラッシュメモリカードのインターフェースを2スロット(メモリースティック，SDカードに対応)装備．PCに搭載することで，メモリースティックとSDメモリカードを同時に使用して，一方から他方に直接データコピーすることが可能．

■ (株)リコー
サンプル価格：R5C592 ¥3,000
R5C593 ¥2,500
TEL：045-477-1703
E-mail：lsi-support@ricoh.co.jp
URL：http://www.ricoh.co.jp/

●DC-DCコンバータ

LM2608/LM2614/LM2618 LM2788/LM2798

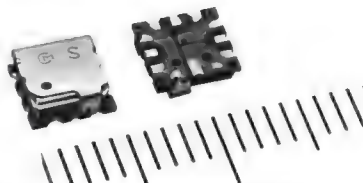
- LM2608，LM2614，LM2618はスイッチングレギュレータタイプのDC-DCコンバータ．最大95%の高効率と±2%の出力電圧精度で，RFやディジタルベースバンドアプリケーションの電源回路向けに高精度ソリューションを提供．
- 出力電流が最大400mAに設定されているため，携帯電話や汎用プロセッサなどのアプリケーションに適する．
- 出力電圧1.8V，1.5V，1.3V，1.05Vでピン選択が可能で，出力電圧設定用に外付け抵抗の必要がない．プロセッサなどが低周波数動作モード時の消費電力削減のため，動作電圧のダイナミックな調整も可能．
- LM2788とLM2798は，外付けのインダクタを必要としない最小のステップダウンコンバータ．2.8V～5.5Vの入力電圧から最大出力電流120mAの低電圧を生成．

■ ナショナル セミコンダクター ジャパン(株)
価格：LM2608/LM2614/LM2618 ¥144(1,000個時)
LM2788 ¥107(1,000個時)
LM2798 ¥119(1,000個時)
URL：http://power.national.com/jpn/

●表面波デュプレクサ

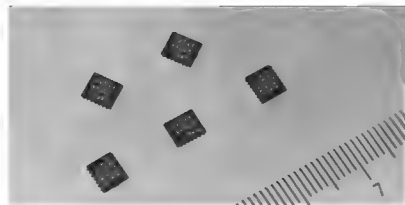
SAYHS836MAA0T00

- CDMA800/TDMA800/E-AMPS端末対応の小型表面波デュプレクサ．
- 新規パッケージの表面波フィルタの採用，小形マッチング素子の搭載により，従来品と比較して面積比で約71%の5.0×5.0×1.85mmのサイズ，約115mgの重量を実現．
- 耐電力は従来品と同等の1.2W/50000時間を保証．
- 減衰特性が優れているため，アンテナとパワーアンプ間の高調波抑圧用ローパスフィルタが不要となり，部品点数の削減にもつながる．
- 受信側フィルタでも，挿入損失，送信側から受信側へのアイソレーション特性において従来品より向上．

■ (株)村田製作所
価格：下記へ問い合わせ
TEL：03-5469-6138

●リチウムイオン電池保護用 IC

S-8254 シリーズ

- ノートパソコンなどの携帯情報機器の電池パックに適する小型，高精度の 3～4 セル直列用リチウムイオン電池保護用 IC.
- センス抵抗を用いた高精度な過電流検出を実現.
- SEL 端子により 3 セル直列/4 セル直列用の切り替えが可能.
- 低消費電流は最大 40μA を実現.
- パッケージは，16 ピン TSSOP（5.1 × 6.4 × 1.1mm）で提供.

■ セイコーインスツルメンツ（株）
サンプル価格：¥150
TEL：043-211-1193

●半導体ディスク装置

PC-SDD V シリーズ

- フラッシュメモリを使用した IDE 接続タイプの記憶装置.
- 大容量 2G バイトモデルをラインナップすることで，Windows2000/NT/98SE/98/95, Linux などの OS をプラットホームとするハードディスクレスのパソコンシステムの構築が可能.
- ハードディスクのような回転，可動部を持たないため，厳しい耐環境性が必要な用途に適する.
- モータの回転音やシークなどのアクセス音がないため，録音スタジオなど静粛性を要求される用途に適する.

■（株）コンテック
価格：¥17,000～¥250,000（IDE 2.5 インチタイプ）
¥20,000～¥253,000（IDE 3.5 インチタイプ）
TEL：03-5628-9286　FAX：03-5628-9344
E-mail：tsc@contec.co.jp

● TFT 液晶パネル

Dream Ⅲ

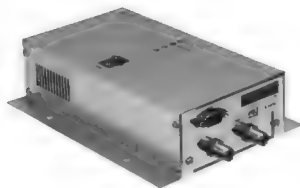
- 高温ポリシリコン TFT 液晶パネル.
- 0.7 型，0.9 型の 2 機種のワイドタイプの液晶パネルを製品化.
- 独自のプロセスルールを採用し，より明るく高精細で色再現性に優れた映像を実現.
- リニアプロジェクタ用途としても使用が可能.
- プロジェクタの軽量化，低消費電力化，低コスト化を実現するコストパフォーマンスに優れたデバイス.

■ セイコーエプソン（株）
価格：下記へ問い合わせ
TEL：03-3340-2637
URL：http://www.epsondevice.com/

●遠隔情報収集装置

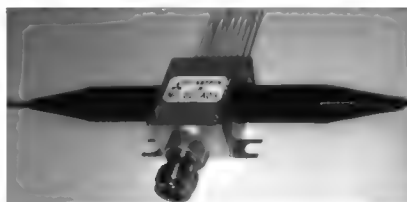
DTU

- 伝送データの品質を補償する機能を搭載した DoPa 対応汎用通信アダプタ.
- TCP/IP プロトコル変換機能，パケット通信機能，データロギング機能をあわせもつ.
- コンパクトフラッシュのスロットを装備し，CPU の搭載，OS に Linux を採用するなど，汎用性の高い装置として利用可能.
- MobileArk と DoPa 対応無線モジュールのどちらでも利用可能.
- ユーザーニーズに合わせたカスタマイズが容易.
- 停電検出時のデータ保存と停電の通知.
- 遠隔からのプログラムやデータのダウンロードが可能.

■ NTT アイティ（株）
価格：¥100,000
TEL：03-3667-8151　FAX：03-3667-8225
E-mail：dtu@fis.ntt-it.co.jp

●半導体光変調器

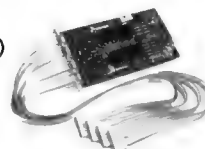
半導体光変調器/半導体光変調器モジュール

- 独自の非対称量子井戸構造を用いることで，高い許容入力光強度と低波長チャーピングを実現.
- 40Gbps 信号の伝送距離を既存の通常分散ファイバ伝送路で 2km 以上，光増幅器のない分散補償ファイバ伝送路で 40km 以上まで拡大可能.
- 基幹系ネットワークでは，光増幅器を挿入して光ファイバ内の光強度の減衰を補償することで，さらに伝送距離を拡大することができる.
- 光変調器に半導体を用いることで，チップサイズを 0.3mm 角にまで縮小でき，大量生産を可能にしている.

■ 三菱電機（株）
サンプル価格：¥200,000（半導体光変調器）
¥800,000（半導体光変調器モジュール）
TEL：0467-41-5207

●ステート/タイミング解析モジュール

16753A/16754A 16755A/16756A

- 同社のロジックアナライザ，16700 シリーズに搭載して使用するステート/タイミング解析モジュール.
- 最大 600Mbps までの高速信号に対応した解析が可能.
- 高速タイミング解析モードである Timing Zoom 機能が 4GHz，64k サンプルに向上している.
- 標準のタイミング解析とステート解析の速度を，従来の汎用モジュールを使った場合と比較して，1.5 倍高速化.
- EyeScan 機能を搭載することで，高速信号で問題となるシグナルインテグリティの問題を多数チャネルでも短時間でアイパターン解析できるようになる.
- 従来と比較して被測定回路への負荷や周波数特性において同等の性能を備えながら，信頼性と操作性を向上.

■ アジレント・テクノロジー（株）
価格：¥2,757,000
TEL：0120-421-345

HARD WARE

●4チャネル拡張PCカード

COM-4(CB)H

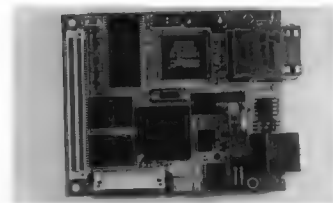
- PC Card Standard 標準 CardBus 対応の PC カード.
- パソコンに挿入することで，4チャネルの RS-232-C シリアル通信ポートを拡張することができる.
- データバス幅32ビット，最大転送速度132Mバイト/sの高速バスであるCardBusをパソコンとのインターフェースに採用したことで，パソコンのパフォーマンスを十分に引き出した多チャネル，高速通信レートのアプリケーションを構築できる.
- 最高921,600bpsの高速通信に対応.
- 各チャネルのボーレートは，ソフトウェアによって個別に設定可能.

■（株）コンテック
価格：¥34,800
TEL：03-5628-9286　FAX：03-5628-9344
E-mail：tsc@contec.co.jp

●CPUボード

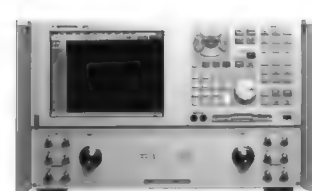
CerfBoardXS

- XScale-400MHzを搭載したCPUボード.
- サーフボックスは，小型(27×108×85 mm)ボックスにWindows CE .NETまたはLinux2.4.xをバンドルし，ネットワーク端末用に最適化されたデバイス.
- インターフェースとして，Ethernet，USB，シリアルポートなどを搭載.
- 遠隔で組み込みデバイスの環境設定や自動化を実現させることのできる製品.

■ 旭テクネイオン（株）
価格：下記へ問い合わせ
TEL：03-5363-8941　FAX：03-5361-8165
E-mail：i-sales@asahi-techneion.co.jp
URL：http://www.asahi-techneion.co.jp/system/

●ネットワークアナライザ

E8361A

- 一体型のネットワークアナライザとして67GHzの上限周波数を実現．代表値としては70GHzまでの測定も可能.
- 信号源，受信部，コネクタを全体に改良することで，測定性能を犠牲にすることなく測定周波数上限を70GHz，また下限周波数を10MHzからに拡張.
- 67GHz時でも93dB以上という広いダイナミックレンジを実現．また，67GHz時でも0.0006dBという低いトレースノイズを実現.
- 従来測定不可能であったフィルタの減衰領域の特性を把握できる.
- 1点の測定あたり26μsという高速測定を実現.
- 24.2×42.5×47.2cmの小型化を実現.

■ アジレント・テクノロジー（株）
価格：¥17,400,000～
TEL：0120-421-345

●EMI簡易測定装置

Common-EMI FC-1000

- 開発段階におけるプリント基板やユニット製品のEMI測定評価，ノイズ対策の効果確認といった分野で活用することが可能.
- EMI簡易測定器として，電波暗室測定と同レベルの性質を有する.
- 電波暗室のうち比較的普及している3m型の設備投資と比較して，導入コストを1/10に低減.
- 卓上型のため，身近に置くことで測定にかかる工程を大幅に短縮することが可能.
- 直接プリント基板に発生するコモンコード電圧を測定するため，余計な放射ノイズの影響を受けにくく，ノイズデータの再現性が良好となり，測定時間も大幅に短縮可能.
- プリアンプと組み合わせて使用することで，バックグラウンドノイズが低い，低ノイズフロアレベルとなる.

■ 富士ゼロックス（株）
価格：¥1,300,000
TEL：046-238-6182

●USB型PHS端末

AH-F401U

- 128kbpsの高速パケット通信を実現したAirH"対応のPHSデータ通信専用端末.
- USB端子接続タイプを採用したことで，本体接続時にパソコンを再起動する必要がない，パソコンの種別を問わず本体への接続が可能，カード型周辺機器の同時使用可能などの特徴がある.
- フレキシブルコネクタと2軸可動設計により，パソコン側USB端子の位置を選ばない.
- DDIポケットが提供する128kパケット方式，32kパケット方式，フレックスチェンジ方式，32kPIAFS/64kPIAFSベストエフォート方式の各種通信方式に対応.
- 小型アンテナを内蔵.
- LEDを2個搭載し，モード表示2色，電波強度2色で通信状態を画面で確認可能.
- パソコンに接続すると同時にユーティリティソフトが起動し，通信/非通信中にかかわらず，電波強度がパソコン画面で確認可能.

■ 富士通（株）
価格：下記へ問い合わせ
TEL：03-3216-8015
E-mail：mobilephone@pc.fujitsu.com
URL：http://www.fmworld.net/

●GPS受信IC

PointChanger SE4100

- カナダのサイジ・セミコンダクタ社が開発したカーナビ/携帯ナビゲーション装置用のGPS受信IC.
- IFフィルタ，VCO，タンク回路，LANを4×4mmの小型パッケージに集積.
- 電源電圧2.7Vからの消費電流が10mA.
- ローカル送信機によるRF過負荷からのクイックリカバリを実現.
- 内蔵LANは，標準1.3dBの低雑音指数を実現する利得切り替えが可能.
- ディジタル出力は4.092MHzで，業界標準のGPSベースバンドソリューションに適する.
- 多数の主要ベースバンド回路と組み合わせることで，連続動作時のシステム全体の消費電力量は120mW以下.

■ コーンズ・アンド・カンパニー・リミテッド
価格：下記へお問い合わせ
TEL：03-5730-1640　FAX：03-5730-1623
E-mail：e-device@tky.cornes.co.jp
URL：http://www.cornes.co.jp/

SOFT WARE

●ソフトウェア開発支援ツール

TestRealTime v2002 Release 2
PurifyPlus RealTime v2002 Release 2
PurifyPlus for Linux v2002 Realease 2

- Rational TestRealTimeは，ネイティブ，組み込み，リアルタイムシステム開発者向けのテストソリューション．C/C++/Ada言語用デバッグ機能を拡張し，デバッグ，テスト，コードの修正を短期間で行うことが可能．
- 新たにJava2がテスト対象言語として追加され，メモリ，パフォーマンスボトルネック，スレッドプロファイリング，コードカバレッジ測定およびランタイムトレースなどの解析機能をJavaプラットホーム上で直接実行することが可能．テスト結果を評価するためのレポートインターフェースも提供．
- Rational PurifyPlus RealTimeは，クロス開発環境におけるメモリ，パフォーマンス，スレッドのプロファイリングやコードカバレッジ測定，UML，ランタイムトレースなどのランタイム解析を行うことができる．

■ 日本ラショナルソフトウェア（株）
価格：Rational TestRealTime v2002 Release 2 ¥924,000
Rational PurifyPlus RealTime ¥420,000
Rational PurifyPlus for Linux v2002 Realease 2 ¥336,000
TEL：03-5642-9160

●FPGA検証ソフトウェア

ChipScope Pro

- ザイリンクスとアジレント・テクノロジーが開発した，ザイリンクスのFPGAのインサーキットデザインを支援するソリューション．
- アジレントトレースコア，およびアジレントFPGAトレースポートアナライザを単一のソリューションにまとめ，R&Dデザインエンジニアがインサーキットデザインのデバッグ時間を短縮し，開発コストを低減し，ザイリンクスFPGA製品の市場投入スピードを短縮できる．
- ザイリンクスISE5.1iに直接組み込むことが可能なオプションソフトウェアツール．
- トレースストレージを備えたケーブルLAN方式のアジレントFPGAトレースポートアナライザは，FPGAの内部コンフィグレーションに高性能なアクセス手段を提供する．

■ ザイリンクス（株）
アジレント・テクノロジー（株）
価格：$695
TEL：0120-421-345

●SoCモデリング動作検証ツール

MaxSim/MaxCore

- C言語でマルチプロセッサシステムレベルの疑似プロトタイプを生成でき，動作検証を行うEDAツール．
- MaxSimは，SoCのシミュレーションモデル生成と動作検証を行うためのツール．プロセッサやDSPが混在したマルチプロセッサ構成のSoCをサイクルアキュレート，またはビットアキュレートで高速に実行可能．シミュレーションモデルとしてはARM, MIPS，DSPGなどのプロセッサ，AMBAバス，DSP，メモリ，および周辺回路など基本的な部品がライブラリ化されている．
- MaxCoreは，CPU，RISC，VLIWなどのプロセッサタイプアーキテクチャのシミュレーションモデルを生成するツール．

■ 丸文（株）
価格：¥7,800,000～
TEL：03-3639-5301　FAX：03-3639-2358

●XMLソリューションパック

Knowledge Publisher
Knowledge Publisher Lite

- Wordを基盤とした独自開発の入力支援ツールにより，XMLドキュメントをより簡便に作成する機能を搭載．
- 通常のWord操作と同様の入力操作と，ダイアログ表示によるスタイル設定でドキュメントの作成が可能．
- 既存のWord文書を取り込む機能により，既存文書の効率的なXML化が可能．
- 目次の自動生成と目次と本文のリンクが可能で，本文を項目別，文節別，章別に自動的に区分けし，見やすい単位で表示，改定情報を画面上で差分表示，利用者への情報通知などの加工を，作成されたXMLドキュメントのタグを活用して自動的に行う．
- データベース管理された情報から，高速検索エンジンにより，テキスト全文検索，カテゴリ検索，XML検索（オプション）などの検索機能を提供．

■（株）富士通ラーニングメディア
価格：Knowledge Publisher Windows版 ¥5,000,000
Knowledge Publisher Solaris版 ¥7,000,000
Knowledge Publisher Lite Windows版 ¥1,000,000
TEL：03-3730-3109
E-mail：sales@flm.co.jp
URL：http://www.flm.fujitsu.com/

●組み込みネットワークプラットホーム

NET+OS 5.0

- Ethernetやインターネット接続が可能な組み込みネットワーク機器に，Webによるリモート管理，モニタリング，ネットワーク上での各種コントロール機能を実現．
- リアルタイムOSや，TCP/IPやUDP, HTTP，FTP，SNMPなどEthernet，インターネット管理システムの開発に必要とされるネットワーキングソフトウェアを含んだ組み込みネットワークソフトウェアパッケージ．
- SNMPv1/2，SNMP MIBコンパイラとソースコードジェネレータを搭載．
- 複数のリソース（HTTPやSNMPなどのプロトコル）が共通のデータにアクセスできる．

■ ネットシリコン ジャパン（株）
価格：下記に問い合わせ
TEL：03-5428-0261　FAX：03-5428-0262
URL：http://www.netsilicon.co.jp/

●技術計算システム

Mathematica 4.2 日本語版

- J/Link2.0と組み込みのJava Runtimeエンジンによる Javaとの透過的な統合を実現．
- XMLの拡張によりノートブックと式がXMLとして保存可能．
- シンボリックXML操作のための新パッケージであるXMLツールを搭載．
- スタイルシートを含む，XHTMLエキスパートをサポート．
- 拡張されたMathML2.0をサポート．
- 線形計画法と最適化を改良．
- 新パッケージANOVAを含む，統計機能を改善．
- 新たな組み合わせ論とグラフ理論のためのパッケージ「Combinatorica」を搭載．
- 技術出版のための新パッケージ「Author Tools」を搭載．
- プレゼンテーションのためのスライドショースタイル環境をサポート．
- FITSとSDTSを含むインポート/エキスポート形式をサポート．

■ ウルフラムリサーチアジアリミティッド社
価格：¥358,000
TEL：03-3518-2880　FAX：03-3518-2877

IPパケットの隙間から

悪徳商法，なぜなくならない？

祐安重夫

悪徳商法については，常日頃からチェックを怠らないようにしている．怪しげな電話には相手の正体が確認できるまで重要な情報を提供しないし，玄関のドアにはセールスと勧誘お断りの表示をしている．テレビのニュースなどでも，悪徳商法についての報道は今までもあったし，最近ではとくに多くなっている．こんな時代に，どういう人間が悪徳商法に引っかかるのかと思っていたら，何と見事に引っかかった例が身近に出現した．

引っかかったのは母の会社で，いくつもの盲点をついたものだった．手口は，

http://www.makani.to/akutoku/bbs/qa/pslg30950.html

の内容そのものである．手口どころか，相手の会社名まで同じだった．

最初は電話がかかってきて，電話機の型番を教えてほしいといってきたというのだが，その際にNTTであるかのように装う表現を使ったらしい．じつはこの会社から，筆者のところにも電話がかかってきたことがあるが，そのときもNTTであるかのような曖昧な表現をしたので，本当にNTTかどうか確認したところ，NTTの代理店だと回答してきた．本当にNTTの代理店かどうかは確認していないが，代理店どころかNTTに対してさえ，電話機の型番を知らせる理由はない．

また，現在販売されている電話機はすべてディジタルだとか，アナログ電話機は今後使用できなくなるとか，とんでもない嘘を並べたてたようだ．結局，アナログ回線にディジタル電話機をつなぐためのインターフェースと，ディジタルビジネスフォンを3台というとんでもない代物のリース契約(7年で総額90万円程度)を，今までのNTTの電話機のリース料とほぼ同額と称して契約させられた．

もちろん，アナログ電話機はいまでも大量に販売されているし，NTTはアナログ電話機が存在するかぎり，アナログ回線のサポートを続けるし，強引にディジタル回線への切り替えを，それも有料で迫ってくることなどない．電話局とうちのビルの間は一部光になっているだろうが(そうでなければISDNやBフレッツは引けない)，アナログ回線も残っているはずだ(現にADSLを引いている)．回線がディジタル化されても，アナログ電話機へ接続するための機器は，NTTがもちろん無料で用意する．

その話を聞いたのは，契約した当日の夜だったのだが，母にはすぐに電話して契約解除を要求するようにいった．どうやら本人はクーリングオフできると思っていたようだが，これは個人で契約した場合にしか通用しない．なまじテレビなどで悪徳商法についての知識を得ていると，こういうところが盲点になる．これについては，

http://www.kobe-np.co.jp/kobenews/sougou/010810ke18290.html

が警告している．

今回はクーリングオフできないので，相手が嘘をついて契約させたというのが，重要な論点になる．いざとなったら，数か月前に別件で仕事を依頼した弁護士を間に立てることも含めて契約解除を迫るように助言したところ，「弁護士」という言葉が功を奏したか，それともこの商法への批判がすでにいくつも出ているためか，翌日の電話で契約解除に持ち込むことができた．さらに翌日には，宅配便で契約書が戻ってきて，無事に一件落着した．

じつは母の会社は筆者も取締役になっているのだが，実質的には70歳をすぎた母と，50代後半の母のアシスタントの人との二人でやっており，どうもハイテクといったわけのわからないものには，結構だまされやすいようだ．実際，そういう事業所を狙い撃ちするように，このような悪徳商法が行われているらしい．

今回はまだリースも始まっていないし，工事さえ行われていない時期に手を打つことができたので，契約解除に持ち込めたが，そうでなければリース会社も含めての訴訟などということになっていたかもしれず，それができずに結局は泣き寝入りというケースは，かなり多いと思われる．さらに多いのは，未だにだまされたと気がついていないケースがあるかもしれないという点が，恐いところである．

電話機についての悪徳商法としては，ISDNが話題になった頃の，強引にISDN契約をさせた事例や，ナンバーディスプレイ導入時に，今後その電話は使用できなくなるといって，電話機を売りつけていた例などがある．マイライン開始時には，各電話会社が，マイラインのために新しい電話機を買う必要はないことを，インターネットをはじめいろいろなところで警告していたのは記憶に新しい．それでも実際に，マイライン契約のために新しい電話機を買わされたユーザーは少なくないような気がする．

この手の悪徳商法は，過去には消防署の方から来ましたといって，消火器を売りつけるものなどがあったが，手口自体は古典的なものである．詐欺や悪徳商法の手口というのは，昔から確立していて，そのネタに何を選択するかということにつきているようだ．とくにハイテク関連をネタに，それがよくわからない老人をターゲットにするなど，ネタの複雑さに単純な手口が隠れてしまっている．

すけやす・しげお インターメディア・アクセス

海外・国内イベント/セミナー情報

INFORMATION

海外イベント

12/3-6 **NEPCON WEST**
San Jose McEnery Convention Center, San Jose, CA, USA
Reed Exibition
http://www.nepconwest.com/

12/9-11 **IEEE International Electron Devices Meeting**
Hilton San Francisco, San Francisco, CA, USA
IEEE
http://www.his.com/~iedm/

12/10-12 **Infosecurity Conference and Exhibition**
Jacob K Javits Center, New York, NY, USA
Reed Exibition
http://www.infosecurityevent.com/App/main.cfm?moduleid=42&appname=100004

1/6-10 **Macworld Conference & Expo**
The Moscone Center, San Francisco, CA, USA
IDG
http://www.macworldexpo.com/macworld2003/V33/index.cvn

1/9-12 2003 **International CES**
Las Vegas Convention Center, Las Vegas, NV, USA
Consumer Electronics Association
http://www.cesweb.org/

1/14-16 **COMDEX SCANDINAVIA**
Swedish Exhibition and Congress Centre, Goteborg, Sweden
Key3Media
http://www.key3media.com/international/events/index.php?d=scandinavia&s=welcome

1/27-30 **COMNET Conference & Expo**
Washington Convention Center, Washington D.C., WA, USA
IDG
http://www.comnetexpo.com/comnetexpo/V33/index.cvn

国内イベント

11/27-29 **Security Solution Expo 2002**
パシフィコ横浜(神奈川県横浜市)
日経BP社
http://expo.nikkeibp.co.jp/secu-ex/

12/4-5 **VON JAPAN**
ヒルトン東京(東京都新宿区)
キースリーメディア・イベント
http://www.von-japan.jp/

12/4-6 **SEMICON Japan 2002**
日本コンベンションセンター(幕張メッセ，千葉県千葉市)
SEMI
http://www.semi.org/japan/

12/4-6 **STREAMING MEDIA ASIA 2002**
東京国際展示場(東京ビッグサイト，東京都江東区)
IDGジャパン
http://www.idg.co.jp/expo/smj/

12/10-11 **MATLAB Expo 2002**
東京ドームプリズムホール(東京都文京区)
サイバネットシステム
http://www.cybernet.co.jp/matlab/me2002/

12/16-20 **Internet Week 2002**
パシフィコ横浜(神奈川県横浜市)
(社)日本ネットワークインフォメーションセンター(JPNIC)
http://internetweek.jp/

1/22-24 2003 **電子コンポーネント EXPO**
東京国際展示場(東京ビッグサイト，東京都江東区)
リードエグジビションジャパン
http://web.reedexpo.co.jp/ed/

開催日，イベント名，開催地，問い合わせ先の順

セミナー情報

最新CCDイメージセンサの特性と技術～CCDの基礎と応用技術～
開催日時　：11月28日(木)～11月29日(金)
開催場所　：CQ出版セミナールーム
受講料　　：25,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125

Linuxデバイスドライバ・デバッグテクニック
――USBデバイスのドライバのデバッグ手法を中心に解説――
開催日時　：11月29日(金)
開催場所　：新百合21研修室(神奈川県麻生市)
受講料　　：45,000円
問い合わせ先：(株)デバイスドライバーズ，☎(042)363-8294
http://www.devdrv.co.jp/seminar/

DSPを使ったシステム開発入門
開催日時　：11月30日(土)
開催場所　：CQ出版セミナールーム
受講料　　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125

MATLABによる制御のためのシステム同定
開催日時　：12月3日(火)～12月4日(水)
開催場所　：オームビル(東京都千代田区)
受講料　　：71,800円
問い合わせ先：(株)トリケップス，☎(03)3294-2547，FAX(03)3293-5831
http://www.catnet.ne.jp/triceps/sem/021203a.htm

PC実習!!　SOAP・WSDL・UDDIプログラミング入門
開催日時　：12月5日(木)～12月6日(金)
開催場所　：SRCセミナールーム(東京都高田馬場)
受講料　　：78,000円
問い合わせ先：(株)ソフト・リサーチ・センター，☎(03)5272-6071
http://www.src-j.com/seminar_no/22/22_254.htm

リアルタイムOS基礎
開催日時　：12月9日(月)～12月11日(水)
開催場所　：東京MIビル26F(東京都品川区)
受講料　　：30,000円
問い合わせ先：三菱電機セミコンダクタ・アプリケーション・エンジニアリング(株)半導体研修センター，
☎(03)5783-7365　http://www.semicon.melco.co.jp/

MPEG-4仕様解説と実装・伝送技術
開催日時　：12月10日(火)～12月11日(水)
開催場所　：オームビル(東京都千代田区)
受講料　　：62,500円(1口で1社3名まで受講可)
問い合わせ先：(株)トリケップス，☎(03)3294-2547，FAX(03)3293-5831
http://www.catnet.ne.jp/triceps/sem/c021210n.htm

移動通信用SAWデバイスの開発と最新技術
開催日時　：12月12日(木)
開催場所　：リアライズ理工学院研修室(東京都文京区)
受講料　　：30,400円
問い合わせ先：リアライズ理工学院，☎(03)3815-8552，FAX(03)3815-8529
http://www.rlz.co.jp/seminar/seminardata.php?id=RGS212007

C言語によるLSI設計～設計事例を通した理論と実際
開催日時　：12月13日(金)
開催場所　：CQ出版セミナールーム
受講料　　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125

組込み向けソフトウェアのテスト・設計技法
開催日時　：12月17日(火)
開催場所　：中央大学駿河台記念館(東京都千代田区)
受講料　　：52,500円(1口で1社3名まで受講可)
問い合わせ先：(株)トリケップス，☎(03)3294-2547，FAX(03)3293-5831
http://www.catnet.ne.jp/triceps/sem/c021217n.htm

ネットワーク/ディレクトリサービスにおけるネットワーク管理法
開催日時　：12月17日(火)～12月19日(木)
開催場所　：高度ポリテクセンター(千葉県，千葉市)
受講料　　：35,000円
問い合わせ先：雇用・能力開発機構 高度ポリテクセンター事業課，☎(043)296-2582
http://www.apc.ehdo.go.jp/seminar/jyohousub_out/syousai/02IIW21210.html

SAN(ストレージ・エリア・ネットワーク)入門技術解説
開催日時　：12月18日(水)
開催場所　：SRCセミナールーム(東京都高田馬場)
受講料　　：48,000円
問い合わせ先：(株)ソフト・リサーチ・センター，☎(03)5272-6071
http://www.src-j.com/seminar_no/22/22_251.htm

バグを出さないための!!ソフトウェア・デバッグ工学とテスト技法
開催日時　：12月19日(木)～12月20日(金)
開催場所　：SRCセミナールーム(東京都高田馬場)
受講料　　：76,000円
問い合わせ先：(株)ソフト・リサーチ・センター，☎(03)5272-607
http://www.src-j.com/seminar_no/22/22_253.htm

FPGAで作るマイクロプロセッサ
開催日時　：12月19日(木)～12月20日(金)
開催場所　：CQ出版セミナールーム
受講料　　：39,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125

日程はすべて予定です．問い合わせ先にご確認のうえ，お出かけください．

読者の広場

2002年11月号特集「徹底解説！　ARMプロセッサ」に関して

▷ハードからソフト事例まで網羅したARMの特集は，非常に役立ちました．実製品でARMを使用したことはまだありませんが，今後のCPU選定の参考にします．（moto）

▷ARMの文字に興味を引かれて購入しました．ARMといえば最近Windows CEとインテルのARMの組み合わせのPDAばかりで，「MIPSがんばれ！」とおもっていたところへ，RAIDカードにDECのDigital印のARMが搭載されていたものを見つけて，「RAIDカードにも使われていたんだ(現行製品にもARMが使われているものある)と意外な一面を知り，ちょっと気になっている製品でした．（A758）

▷ARMプロセッサを使ってみてもよいのではないかと，少し興味が湧いてきた．ただ，少し気がかりなのは，CPUとして決まったスペックのハードウェアがないという点で，コストも含めて時期尚早という感じもします．（Yahoo!）

▷いままで，RISCプロセッサで組み込みといえばSuperHしか浮かんできませんでしたが，本号でARMアーキテクチャの魅力を発見しました．とてもためになりました．（ビギナーズ）

▷今回，最初に工場の作業の例でプロセッサの動作を説明している文中，敷地外の製品倉庫(データメモリ)から作業棚(レジスタ)に移すには～以下の説明は，あまりにも解説が少なすぎて，いいかげんなように思った．また，ARMプロセッサを実験で理解できないものか！　ICおよびテスト基板などを調べようと思ったら，いきなりコラムで秋葉原などでは市販されていないとか．せめて，どうすれば実験(体験)できるのか，具体的に方法を書いてくれても良いのではないだろうか．まるで他人事のような書き方である．（とくめい希望）

その他

▷何かと耳にするARMだが，どのようなものかを知るには十分な内容だった．「やり直しのための信号数学」は，おもしろそうだが，ひさしぶりに購入したので理解不能．単行本化を望みます．（KAZZU）

▷組み込み系の会社から内定をいただいて，本誌を熟読するようになりました．ジャイロとGPSを使った会社なので，ARMやMotionJPEGなどが将来結びつきそうでわくわくしました．これからも宜しくお願いします．（保坂欽大）

▷久々に掲載された「組み込みプログラミングノウハウ入門」でしたが，楽しく読ませてもらいました．後半でオブジェクト指向の話も少し触れられていることもあり，今後ステートマシンとオブジェクト指向の現実的なつなぎ合わせ(CASEツールを絡めて)の話題もあるものと期待しています．（は）

▷「シニアエンジニアの技術草子」にあったWebブラウズ，mail，OfficeができるPCは，LinuxやFreeBSDの上にMozillaとOpenOfficeを乗せ，セキュリティパッチをDebianのapt形式であてるようにすれば，家電メーカーならどこでも作れそうですよね．どこか作らないかな……（玉出のタマ）

▷「フジワラヒロタツの現場検証」の内容は，恵まれた才能をもつ人だからこそ，となりの芝生が青く見えるのであって，となりの家という選択肢が存在しない人も多いはず．いろいろな体験(＝苦労？)をできるというのは，とても素晴らしいことだと思います．（クルルンソファ）

▷本誌とは直接関係ありませんが，現役のエンジニアがノーベル賞を受賞し，同じエンジニアとしてたいへんうれしい限りです．同じエンジニアを讃える番組(？)の「プロジェクトX」は少し鼻につきますが，今回の受賞は手放しで喜べます．（慈恩）

特集担当デスクから

☆CPUに内蔵されている/チップセットが市販されている/秋葉原で格安に売られている……そのような状況にもめげず(?)，あえてFPGAで設計する意味がどこにあるのか．その答を今回の特集で理解していただけるよう，筆者といっしょにがんばったつもりですが，いかがだったでしょうか．

☆あぁ～それにしてもページが足りない！　普通はそれぞれのバスやインターフェースだけでも，特集が1本組めるのに，それらをまとめて1回で……というのは，やはり無理があったかもしれません．とはいえ，バラ売り(?)ではインパクトに欠けるのも事実．ホストCPUとメインメモリ，拡張バス，そしてそのバス上にグラフィックス表示にキーボード＆マウス，そしてHDDくらいつながらないと，パソコンとはいえないでしょう！

☆本当は，PCカード/CFカードコントローラや，光ディジタルIN/OUT対応のPCMサウンドカードなども設計しているのですが，誌面の都合で今回は解説記事を掲載できませんでした．いずれ機会があれば，掲載していきたいと考えています．

☆さて，ハードという器ができたらその次はソフトです．このハードウェア上に，ITRONやLinuxを移植していただける方を大募集しています!!

読者の広場

アンケートの結果

興味のあった記事（2002年11月号で実施）

①第1章 ARMアーキテクチャ詳解
②第2章 ARM命令セットの詳細
③第3章 ARMプロセッサを採用したシステムの最適化
④第4章 ARMプロセッサプログラミング事例解説
⑤JPEG2000/Motion-JPEG2000の技術概要と応用（前編）
⑥ハッカーの常識的見聞録（第23回）
⑦組み込み向けGUIフレームワークとIDE（後編）
⑦やり直しのための信号数学（第13回）
⑨移り気な情報工学（第29回）
⑩Appendix ARM搭載評価ボードのいろいろ
⑪フジワラヒロタツの現場検証（第64回）
⑪シニアエンジニアの技術草子（弐拾壱之段）
⑬GUIクロスプラットホームアプリケーションの設計技法
⑭無線LAN専用セキュリティシステム「C4-SWL-BOX」の概要
⑭MEDIA Encoderスクリプトの再利用
⑭組み込みプログラミングノウハウ入門（第5回）
⑰WIRELESS JAPAN 2002
⑰InterGiga No.28
⑲第5章 ルーレットプログラムの作成事例
⑳開発環境探訪（第13回）

特集「徹底解説！ ARMプロセッサ」についてのアンケートの結果報告

Q1　CPUにARM系を採用したことがありますか？

①ある（93%）
②ない（7%）

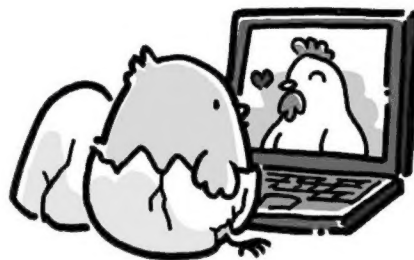
Q2　（Q1で1と答えた方にお伺いします）ARMを採用したシステムで，具体的なCPUコアは何を使われましたか？

① ARM7TDMI（100%）
② ARM72x/74x（0%）
③ ARM9系（0%）
④ ARM10系（0%）
⑤ StrongARM系（0%）
⑥ XScale系（0%）
⑦その他（0%）

Q3　（Q1で1と答えた方にお伺いします）ARM系CPUを採用した理由は何でしょうか？

①低消費電力だから（0%）
② IPとしてSoCに組み込めるから（100%）
③使いなれているから（0%）
④その他（0%）

Q4　ARMに関連した分野で，どのような記事をご希望ですか？

SoC組み込みの注意点など．PDAに使用されWindowsCEやLinuxが動作するARM．PDAアプリケーション開発のためのツールおよび開発手法．応用事例，活用例の詳細および方法論など．

次号予告 ワイヤレスネットワーク技術入門

2003年2月号は12月25日発売です

IEEE802.11系無線LAN技術/標準化動向/Bluetoothプロトコルスタックの開発・検証/OFDM無線モデムの基礎技術と設計事例/FFT計算/60GHz帯を使った高速無線伝送/ミリ波自己ヘテロダイン伝送方式/UWB技術/

IEEE802.11a/b/g，Bluetooth，UWBなど，ワイヤレスLANの規格が乱立状態になっている．そこでまず，これらの規格に使われている技術を原理から解説し，わかりやすく整理する．次に，Linux対応のBluetoothプロトコルスタックについて，サンプルアプリケーションの作成例も示しながら解説する．そして，無線LANなどで使われるディジタル変調方式OFDM(直交波周波数分割多重)について，その原理から，OFDM無線モデムの設計手法までを徹底解説する．また，映像などの大容量伝送に向くといわれる60GHz帯の無線伝送技術について，周波数安定性の問題を解決するための「ミリ波自己ヘテロダイン伝送方式」を中心に説明する．最後に，IEEE802.11やBluetoothに比べ低消費電力，高速伝送も可能なUWB(Ultra Wideband)について，原理や最新状況を解説する．

★次号には，記事関連ファイルなどが満載されたCD-ROM『InterGiga No.29』が付属します！

編集後記

■拉致問題，劇場テロ，狙撃事件，……読むのにつらいニュース記事が続いている．「……Imagine there's no Countries(想像してごらんよ，国境のない世界を)……」とJohn Lennonが歌っている．人は酷いこともするけれど，素晴らしいことも成し遂げる——不思議．いまの状況を良くすべく，自分のベストを尽くすだけ．(洋)

■京都に出張した際，駅の近くに「手塚治虫ワールド」があるので，入ってみた．場内のおみやげ品売り場は手塚治虫のキャラクターグッズで溢れていて，いろいろ目移りしたのだが，家族への土産としてクッションを一つ買った．帰宅後，土産は家族から強い非難を浴びることになった．「ヒョウタンツギ」のデザインだったのだ(^^;)．(= IO)

■今月号のプロローグ……実際には1晩で決まったわけではく，何度も何度も検討を重ね激論を交わした末に決めた仕様です．ついにその全貌(掲載できなかった部分もありますが)を公開できる時が来ました！(しみじみ)　しかしまぁ～会話の端々から，未だに強力なモトラリアンなオーラが感じられる筆者さんです(^^;)(M)

■Chiptuneという「一つの音源チップで完結した音楽」を扱ったパーティーに行ってきました．コモドール64で使われた6581(SID)などの8ビット時代の音源チップで最新の音楽が作られているということで，聞いてみれば有名なあの曲も6581製ということで驚きました．古い技術を新しく使う，というのは痛快ですね．(み)

■田中耕一さんのノーベル賞はうれしいニュースだったが，このような発見でノーベル賞をもらえたことに驚いた．今後の遺伝子工学などに対する影響の大きさが評価されたのだろうが，ひょっとしたら自分もという人も多いのではないだろうか．ソフトウェアに関してノーベル賞が用意されていないのが残念だ．子供でも知っている賞の存在は大きいと思う．(Y)

■自宅から富士山が見えるところに住んでいるので，白く衣替えした富士山を見て冬が近づいてきたんだと感じます．今年は急に夏から秋へと替わり，気が付けば紅葉が始まっていたという感じですよね．この調子できっと冬も来てしまうのでしょう．また寒い季節が始まったんですね．(Y2)

■10月も終わりでいよいよ本格的に寒くなってきました．あったか～い炬燵でみかんでも食べながら好きなテレビを見る．幸せのひと時……そのために大変な出費をしました．110度CSアンテナチューナー，新しいテレビ，月々の視聴料．あとね，ビデオにするかDVDにするか思案中．元採るまで見るのはシンドイだろうな～．(ま)

■携帯電話を買い換えました．携帯を持ち始めて6～7年になりますが，これで3台目．1台を約3年は使ってしまうため，換える度に機能の進化にビックリします．そして毎回，最初はものめずらしくて嬉しいのですが，使う機能はすぐに限定されてしまいます．CMなみに楽しんでるユーザーって全体のどれくらいなの？(M-o)

お知らせ

▶読者の広場
本誌に関するご意見・ご希望などを，綴じ込みのハガキでお寄せください．読者の広場への掲載分には粗品を進呈いたします．なお，掲載に際しては表現の一部を変更させていただくことがありますので，あらかじめご了承ください．

▶投稿歓迎
本誌に投稿をご希望の方は，連絡先(自宅/勤務先)を明記のうえ，テーマ，内容の概要をレポート用紙1～2枚にまとめて「Interface投稿係」までご送付ください．メールでお送りいただいても結構です(送り先はsupportinter@cqpub.co.jpまで)．追って採否をお知らせいたします．なお，採用分には小社規定の原稿料をお支払いいたします．

▶本誌掲載記事についてのご注意
本誌掲載記事には著作権があり，示されている技術には工業所有権が確立されている場合があります．したがって，個人で利用される場合以外は，所有者の許諾が必要です．また，掲載された回路，技術，プログラムなどを利用して生じたトラブルについては，小社ならびに著作権者は責任を負いかねますので，ご了承ください．

▶コピーサービスのご案内
本誌バックナンバーの掲載記事については，在庫(原則として24か月分)のないものに限りコピーサービスを行っています．コピー体裁は雑誌見開きの，複写機による白黒コピーです．なお，コピーの発送には多少時間がかかる場合があります．

● コピー料金(税込み)
1ページにつき100円
● 発送手数料(判型に関わらず)
1～10ページ：100円，11～30ページ：200円，31～50ページ：300円，51～100ページ：400円，101ページ以上：600円
● 送付金額の算出方法
総ページ数×100円＋発送手数料
● 入金方法
現金書留か郵便小為替による郵送
● 明記事項
雑誌名，年月号，記事タイトル，開始ページ，総ページ数
● 宛て先
〒170-8461　東京都豊島区巣鴨1-14-2
CQ出版株式会社　コピーサービス係
(TEL：03-5395-4211，FAX：03-5395-1642)

▶お問い合わせ先のご案内
● 在庫，バックナンバー，年間購読送付先変更に関して
販売部：03-5395-2141
● 広告に関して
広告部：03-5395-2133
● 雑誌本文に関して
編集部：03-5395-2122
記事内容に関するご質問は，返信用封筒を同封して編集部宛てに郵送してくださるようお願いいたします．筆者に回送してお答えいたします．

Interface
©CQ出版(株)　2003　振替　00100-7-10665
2003年1月号　第29巻　第1号(通巻第307号)
2003年1月1日発行(毎月1日発行)
定価は裏表紙に表示してあります

発行人／蒲生良治
編集人／相原　洋
編集／大野典宏　村上真紀　山口光樹　小林由美子
デザイン・DTP／クニメディア株式会社
表紙デザイン／株式会社プランニング・ロケッツ
本文イラスト／森　祐子　唐沢睦子
広告／澤辺　彰　中元正夫　渡部真美

発行所／CQ出版株式会社　〒170-8461　東京都豊島区巣鴨1-14-2
電話／編集部(03)5395-2122　URL http://www.cqpub.co.jp/interface/
広告部(03)5395-2133　インターフェース編集部へのメール
販売部(03)5395-2141　supportinter@cqpub.co.jp
CQ Publishing Co.,Ltd.／1-14-2 Sugamo，Toshima-ku，Tokyo 170-8461，Japan
印刷／クニメディア株式会社　美和印刷株式会社
製本／星野製本株式会社

日本ABC協会加盟誌
(新聞雑誌部数公査機構)　ISSN0387-9569

Printed in Japan